(2)

RL-TR-92-118
Final Technical Report
May 1992

VERIFICATION AND VALIDATION OF AI SOFTWARE

Advanced Decision Systems

R.A. Riemenschneider, Theodore A. Linden, Karen Morgan,
William Vrotney

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-92-118 has been reviewed and is approved for publication.

APPROVED:

Jennifer D. Skidmore

JENNIFER D. SKIDMORE, ILT
Project Engineer

FOR THE COMMANDER

John A. Graniero

JOHN A. GRANIERO
Chief Scientist for C3

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL(C3CA) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May 1992	Final Sep 89 - Sep 91

4. TITLE AND SUBTITLE
VERIFICATION AND VALIDATION OF AI SOFTWARE

5. FUNDING NUMBERS
C - F30602-89-C-0201
PE - 65502F
PR - 3005
TA - RB
WU - 71

6. AUTHOR(S)
R. A. Riemenschneider, Theodore A. Linden, Karen Morgan, William Vrotney

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Advanced Decision Systems
1500 Plymouth Street
Mountain View CA 94043-1230

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory (C3CA)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-92-118

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: 1Lt Jennifer D. Skidmore/C3CA/(315) 330-4031

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This document provides practical advice on how to improve V&V on Al projects. The question we attempt to answer is: How can I apply my knowledge of V&V practice to Al development, which seems very different from the examples from textbooks, and which cannot be easily mapped into the lifecycle models of the DOD standards?

Part I lays a firm foundation by defining terms such as verification, validation, and artificial intelligence. Also, a new representation of system lifecycles is presented which we believe you will find useful in analyzing your organization's Al development efforts.

In Part II the focus shifts to providing advice, with section addressed to project leaders, system specifiers's, designers, programmers, and documenters. Each role contributes in a different way to the overall V&V process, so we present a set of guidelines specific to each role.

Part III is a collection of three appendices: (1) A user's manual for a software tool, ASP, developed under this contract which supports the V&V process by allowing programmers to better integrate formal testing with code development; (2) A glossary of V&V terms; and (3) A guide to commercially available CASE tools.

14. SUBJECT TERMS
Artificial Intelligence, Software Verification, Validation, V&V Testing, Debugging, Software Tools

15. NUMBER OF PAGES
146

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

CONTENTS

1. Introduction

The goal of this document is to provide practical advice on how to improve V&V on AI projects. While there is some discussion of the state of the art in V&V research—and some attempt to advance the state of the art—the emphasis is on guidelines and tools that can provide immediate help in present day development efforts. We assume that the reader is familiar with standard V&V practice, as reflected in textbooks (e.g., [9, 45]) and DoD applicable standards, such as 2167A and 2168. The question we attempt to answer is: *How can I apply my knowledge of V&V practice to AI development, which seems very different from the sorts of examples discussed in textbooks and which cannot be easily mapped into the lifecycle models of the DoD standards ?*

In Part I, the emphasis is on laying a firm foundation. Terms that play a central role in subsequent discussion—including 'verification,' 'validation,' and 'Artificial Intelligence'—are defined, to reduce the chances of misunderstanding our advice. Also, a new representation of system lifecycles is presented, which we believe you will find useful in analyzing your organization's AI development efforts.

In Part II, the focus shifts to providing advice. The part consists of five sections, one addressed to project technical leaders, one to system specifiers, one to system designers, one to programmers, and one to documenters. Each role contributes in a unique way to the overall V&V process, and so we present a set of guidelines specific to each role.

Part III consists of a collection of three appendices. The first is the User's Manual for a software tool, ASP, we have developed to support the V&V process by allowing programmers to better integrate formal testing with code development. Our hope is that the tool increases the value of formal testing so much that programmers will see it as a help rather than a burden. The second appendix is a glossary of V&V terms. The final appendix is a guide to commercially available CASE tools.

Part I

V&V AND AI

2. What V&V Is

"V&V" is short for "verification and validation." The objective of this chapter is to guarantee that we have a common understanding of just what the words "verification" and "validation," mean as they are used in the report. "Verification" in particular is used in a number of different ways by software developers. (Some debates about the utility of verification in AI software development could be settled simply by having the participants explain what they meant by "verification".)

Verification and validation are complementary software development activities, each contributes to software quality. *Verification* is the process of assuring internal consistency in software development, while *validation* is the process of assuring that the software being developed satisfies its requirements. A popular way of saying this is

> *Verification assures that the software is built right; validation assures that the right software is built.*

One important form of verification is assuring that the various products generated during software development process are consistent. Let us call this *inter-product verification*. In virtually every effort to develop software for use by someone other than the author, there are multiple products of the development. At a minimum, there is the software and some description of how to use it. In this case, *inter-product* verification amounts to making sure that the description correctly (and completely) describes the interface to the delivered software. But typically, especially in large development efforts, there are many more products of software development. In addition to the software itself, these may include

- a statement of the requirements that the software must satisfy,
- a specification of properties of the software system—its functionality, real-time behavior, and so forth,
- a design document, or even multiple design documents at varying levels of detail and employing a variety of design formalisms,
- source code for the software,
- a description of test procedures and test cases, at both the system and subsystem level,
- additional software that supports testing,
- a user's manual, and
- a maintainer's manual

Thus, inter-product verification often requires making sure that a system with the specified properties will satisfy the stated requirements, that the design correctly elaborates the specification, that the source code correctly implements the design, and so on.

A second form of verification is *intra-product verification*, assuring that each product of software development is internally consistent. A canonical example from AI is validation of a knowledge base. If a knowledge base consists of logical formulas that are intended to be true of the application domain, that knowledge base should be logically consistent, for a logically inconsistent set of formulas must contain at least one formula that is false. Moreover, since any formula whatsoever can be derived from an inconsistent set of formulas by correct reasoning, the usual justification for accepting the conclusions derived from the knowledge base—that the inference mechanism reasons correctly—has been undermined. A system with an inconsistent knowledge base is generally useless, so knowledge base verification includes assuring consistency. It may also include assuring that the knowledge base has other properties—say, that none of the formulas can be derived from others—judged to be desirable. The consistency criterion can be applied to most lifecycle products: a requirements statement should be consistent in the sense that there is no property that it both requires and forbids the system to have; a system specification should be consistent in the sense that there is no property that it guarantees the system will both have and lack; and so on. Just as in the case of the knowledge base, there are usually other acceptability criteria for the products that must be assured as well. A typical example is assuring that source code meets applicable coding standards. The dividing line between intra-product verification and inter-product verification is somewhat arbitrary—if the design is documented as a series of successive refinements, checking that a purported refinement is in fact a refinement is intra-product verification if the entire series is thought of as a single product, but is inter-product verification if each design in the series is thought of as a product—but useful in practice, especially on larger projects where different products are produced by different groups of developers.

A common error in discussions of V&V is to confuse verification against a statement of requirements and validation against the true requirements. Generally, software is written to solve some problem, and the people who create the software that is intended to solve the problem are not the people who have the problem. To be successful, the software developers must solve the problem, but they must also satisfy the terms of the contract, formal or informal, with their clients. If the clients have a good understanding of what they want in a solution, the contract might contain a statement of requirements. This is the sort of case where it is particularly tempting to say that validation consists of making sure the software satisfies the stated requirements. But it is still important to distinguish between the stated requirements and the actual requirements, because satisfying the stated requirements *may* not be enough guarantee that the problem will be solved. Ideally, validation consists of using the system to actually solve the problem, or at least representative instances of a general problem. If this is impossible, weaker forms of validation, such as running the system in some

sort of testbed environment on simulated data, may have to suffice. When the requirements statement is an input to the development process rather than a product of the development process, checking that the stated requirements are satisfied cannot be considered to be verification. It can be considered validation, but—here is the main point!—it is a very weak form of validation. A system cannot be *well* validated simply on the basis of *stated* requirements, because discovering and correctly stating (at a low enough level of detail that satisfaction can be effectively determined) all the requirements for solving a problem is tremendously difficult. For the problems typically addressed by AI, producing a "completely adequate" statement of requirements is impossible, from a practical point-of-view.

Validation of a system can only be achieved by "trying it out." Other development products can be validated to a limited extent, by trying them out—for example, an executable functional specification can be validated by using it as a prototype, and a user's manual can be validated by having a prospective user attempt to run the system by referring to the manual—or indirectly in conjunction with verification. In fact, a common method of validating a requirements statement is to develop software that has been verified to (at least partially) implement it, and then validating the prototype through use. Thus, an attempt to validate the software system by reference to those stated requirements has the matter reversed. Once the system is sufficiently complete that direct validation though use is possible, the requirements statement should play no further role in system validation, because any conflict between the stated requirements and the system should be arbitrated by direct validation against the true requirements, since, in most cases, the stated requirements are as likely to be wrong as the software. The bottom line is: If the system solves the problem, it doesn't matter that the requirements statement was wrong. (Of course this does not mean that the requirements statement need not be corrected. The requirements statement plays an important role during system maintenance, since it provides guidance on what can and cannot be changed. *It should always represent the best current* understanding of the true requirements.) So, while the system may derive significant indirect validation during development by virtue of its verified satisfaction of the partially validated requirements statement, the fundamental system validation activity—direct validation of the system through use—provides further indirect validation of the stated requirements. That is, once the system is running, the system is used to validate the requirements statement rather than the reverse.

3. What AI Is

"AI" is short for "Artificial Intelligence." So far, so good. But, judging from the literature, few people have ever agreed on a definition of "Artificial Intelligence." For our purposes, it is most convenient to adopt a version of the position that AI is what AI people do. Since our concern is with software systems, this amounts to identifying AI with a type of program, namely, those which use the programming *techniques* used in paradigmatic AI programs, embedded in an *architecture* characteristic of paradigmatic AI programs.

This position proves convenient because the only features of a program that much influence the choice of V&V methods is the collection of programming techniques employed and the architecture. Many other factors are relevant to the choice—ranging from the type of problem addressed, through the lifecycle model used to guide the development, to contractual obligations—but these are external to the program. It turns out that these other factors are, effectively, less relevant to the choice if we restrict our attention to AI software, since most AI systems address problems that are similar in the relevant respects, most AI development efforts use lifecycle models that are similar in relevant respects, and so on. So in this Chapter, we will enumerate some of the programming techniques that make a system AI, categorize some AI architectures, and then look at the common factors in AI system development that influence the choice of V&V methods. Finally, we comment on the potential for using AI techniques to support V&V of AI systems.

3.1 AI Programming Techniques

The following list was largely extracted from a standard text on AI programming techniques, Charniak, et al.'s *Artificial Intelligence Programming* [13], a good overview of the subject. (Another good source on the subject is Norvig's *Paradigms of AI Programming* [33].)

3.1.1 Data-driven Programming

Attaching programs to data and deciding what to do by retrieving and running programs associated with data is *data-driven programming*. One common example is use of message passing to invoke methods associated with the message. As this example illustrates, data-driven programming is not restricted to AI. However, it tends to be especially common in AI, because AI programs are often written in languages, such as LISP and PROLOG, that allow programs to be treated as data. This capability facilitates storing code in comparatively complex data structures, such as a-lists and hash tables, and so encourages use of data-driven techniques. (Another reason for including this technique in our list is that use of data-driven programming strongly influences the choice of V&V methods.)

3.1.2 Discrimination Nets

A common programming problem in AI is the classification of information based on tests of its properties. An initial test is applied; based on the result, another test may be chosen and applied; based on the result, another test may be chosen and applied; and so on. Thus, the collection of tests can be thought of as defining a network, with a link from one test to another whenever the result of applying the former can trigger the application of the latter. Such a network is a *discrimination net*. A good example of applying this technique to an AI problem can be found in Appendix A, which describes an AI-based V&V tool developed at ADS that employs a special-purpose language for defining discrimination nets for software testing.

3.1.3 Meta-level Control Structures

A common form of data-driven programming in AI is the use of a closure to represent the state of a suspended process. A collection of loosely related techniques—*agenda-based control, queue-based control, streams, coroutines, possibilities lists*, and so on—have been based on this idea. The approach is generally useful when meta-level reasoning about control is performed. Object-level operation (which may be search of a game tree, adding information to a blackboard, refining a plan, or just about any other typical object-level activity) is suspended, a "next move" is determined, and the suspended operation is then resumed. LISP includes a variety of constructs that support this technique, including COMMON LISP's closures and SCHEME's first-class continuations, and expert system shells and other higher-level program development tools frequently support some form of meta-level reasoning via this technique.

3.1.4 Deductive Information Retrieval

Many AI applications derive information from facts stored in some sort of knowledge base. Such applications can be thought of as "smart" databases, capable of retrieving not only information that has been stored in them explicitly, but also of retrieving information implicit in the stored facts. Such systems are said to perform *deductive information retrieval*. "Deduction" is used rather broadly here, to include not only strict logical deduction, or even non-monotonic inference procedures modeled on logical deduction, but also the ad hoc heuristic procedures used in semantic networks. Deduction, in this generic sense, is any inference procedure applied to the explicitly represented information.

It may seem that the V&V methods appropriate to logical deduction and those appropriate to looser forms of inference would be quite different, making further subdivision of this technique useful. However, in practice, even strictly deductive inference procedures are incomplete—and sometimes even unsound—theorem provers, for the sake of efficiency. To pick the most widely known example, standard PROLOG employs an

unsound algorithm for unification; the so-called "occurs check" is omitted in order to make unification of a term with a variable $O(1)$ rather than $O(n)$ in the length of the term, just as assignment is. Without this deviation from "logical purity," PROLOG could not compete in efficiency with conventional languages. A clever programmer can easily arrange things so that no incorrect conclusions are derived, and, clearly, this is exactly the sort of thing that V&V procedures should check. Therefore, rather than attempting to subdivide the class of deductive information retrieval techniques, we will focus on assigning V&V techniques to these systems based on properties that the inference method is intended to possess.

3.1.5 Production Systems

A *production system* consists of a collection of condition-action rules, the *production memory*, and a data store, the *working memory*. Its operation is controlled by a *recognize-act loop*: a rule whose condition is true is found, and the corresponding action is performed. The action generally includes changes to the working memory that influence which conditions are satisfied on the next cycle. Thus production systems use a natural generalization of forward-chaining inference. A number of widely-used expert system tools, such as OPS5, support building production systems.

3.1.6 Frame Databases

A *frame* is a data structure used for knowledge representation that naturally generalizes record structures and LISP's property lists. According to Minsky [31],

> [w]e can think of a frame as a network of nodes and relations. The "top levels" of a frame are fixed, and represent things that are always true about the supposed situation. The lower levels have many terminals—"slots" that must be filled with specific instances or data.

Frames have been specialized for particular purposes; Shank and Abelson's *scripts* [37], used for natural language understanding, are a good example of specialized frames.

Just as in the case of production systems, frame databases provide a general purpose knowledge representation that is the basis of several popular expert system building tools, such as KRL and FRL. (Frames and production rules combine in a natural fashion, and many expert system shells provide both techniques.)

3.1.7 Backtracking

Search, in one form or another, is central to AI. The most common search paradigm is to proceed *depth-first*, that is, to attempt to find a path through the search tree

starting from the top and exploring a single branch at a time. For example, a planning system might attempt to refine a plan by selecting among various refinement operators, choosing the operator according to some principle ranging from the simple and inexpensive (such as choosing the first applicable operator from the list of operators) to the complex and costly (such as calculating some measure of which operator is "best" among all applicable operators based on detailed consideration of the current state of the plan). Domain-specific knowledge—sometimes represented explicitly, and sometimes implicitly as an operator ordering or a numerical formula used in computing an operator quality metric—is used to guide the search. If choosing the wrong branch of the search tree (e.g., choosing the wrong refinement operator) can lead to a dead-end, some form of *backtracking* is employed.

In the simplest case, *chronological backtracking*, the most recent decision is changed: state changes associated with actions performed as a result of that decision are undone, and a different decision is made, resulting in the exploration of an alternative branch of the search tree. But if the system tracks the reasons for each decision and can analyze the cause of failure, a more intelligent form of backtracking, *dependency-directed backtracking*, can be employed. Rather than backtracking to the most recent decision, which may have been irrelevant to the failure, control passes back to a decision point that is certainly relevant. Moreover, the system may use sophisticated *reason maintenance facilities* to avoid having to undo all state changes associated with the failed branch; some of the decisions made after the decision being reconsidered may be essentially independent of that decision, and so need not be reconsidered. This might be thoughts of as a jump across the tree rather than "backtracking;" see Fig. 3-1.

The impact of particular programming techniques on choice of V&V methods is discussed in Section 8.

3.2 AI Programming Tools

AI programming tools span the range from programming languages especially suited to AI programming, through high-level programming environments, to knowledge-based system shells. Choice of tools has some effect on appropriateness of V&V methods, so we will briefly summarize some relevant features here to prepare for the discussion of the impact of tool choice on V&V in Section 5.

3.2.1 AI languages

LISP is really a family of programming languages, based on a common core of ideas, that has been the primary vehicle of AI research for over 30 years. LISP was designed for symbolic, rather than numerical, programming, and has grown over the years to include precisely those features most useful for AI programming. For example, LISP has a flexible type mechanism that makes it easy to alter and expand data

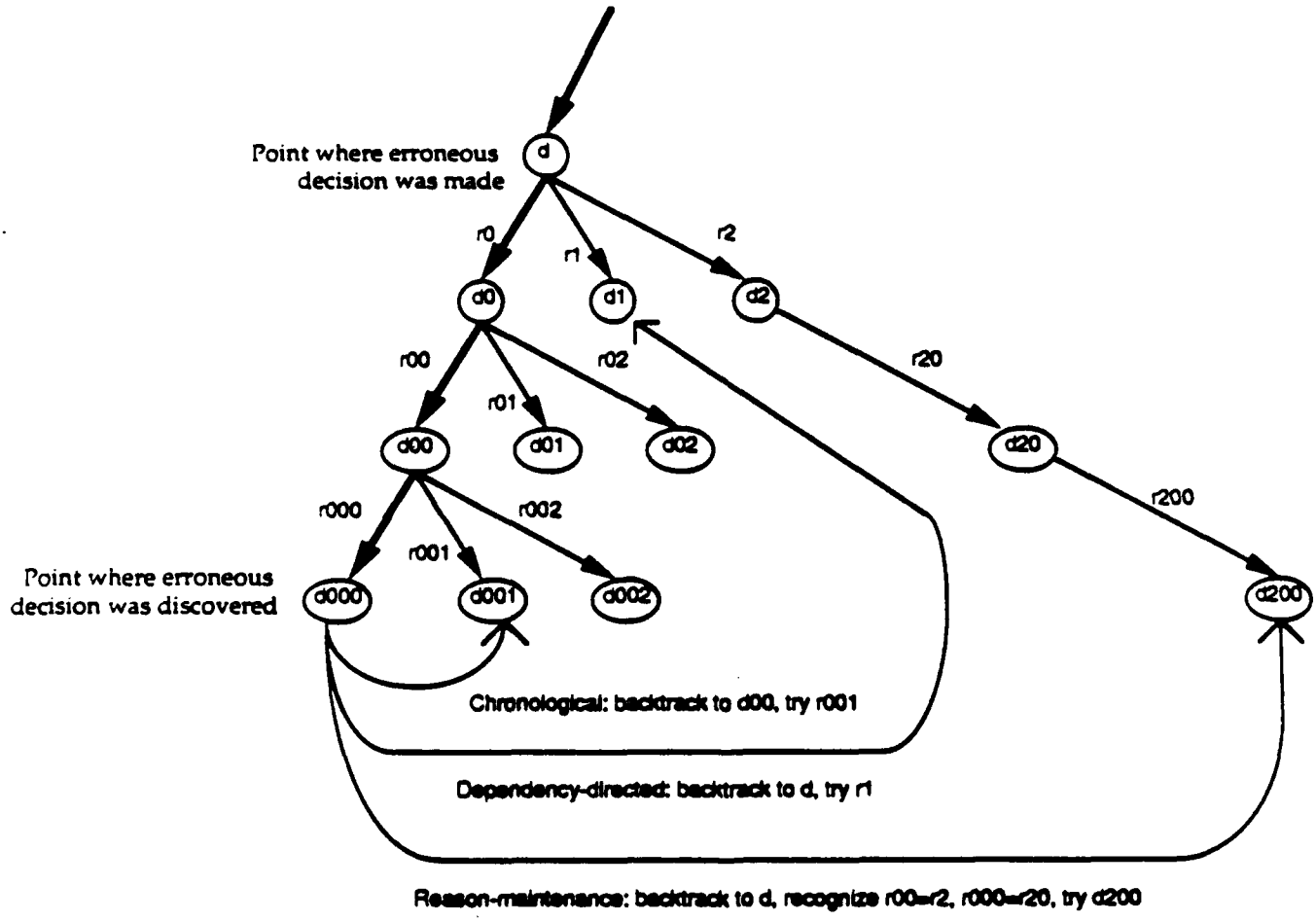

Figure 3-1: Varieties of Backtracking

representations, a feature of obvious utility in incremental development of prototype systems. The fact that programs are represented as data structures makes it easy to develop "meta-level" facilities, including program development tools. Every reader of this manual is probably familiar with the basics of LISP. The best source for a detailed explanation of why LISP is special, and especially suited to AI, is Allen's *Anatomy of LISP* [2].

Compared to LISP, PROLOG is a newcomer on the AI scene, being only about 20 years old. PROLOG was a first attempt at realizing the ideal of logic programming, i.e., treating computation as deduction. PROLOG can be thought of as an efficient constructive theorem prover for a subset of first-order logic. As a result, PROLOG has a firm logical foundation to support the analysis of programs, in rather sharp contrast to LISP. But, more simply, PROLOG can be thought of as the result of supplying the *program clause*

$$P(x, y) \text{ if } Q_1(x, z_1) \text{ and } Q_2(x, z_2) \text{ and } \ldots \text{ and } Q_n(x, z_n).$$

with a *procedural reading*:

> For any $x, y, z_1, z_2, \ldots, z_n$, the goal of proving $P(x, y)$ can be satisfied by proving $Q_1(x, z_1)$, $Q_2(x, z_2)$, ..., and $Q_n(x, z_n)$.

and using a depth-first search strategy based on the ordering of program clauses and chronological backtracking to find a reduction of a given goal to formulas stored in a knowledge base. Sterling and Shapiro's *The Art of Prolog* [42] provides a nice mix of the theory and practice of PROLOG.

The claim that PROLOG was a reasonably efficient general purpose programming formalism was met with considerable scepticism in the United States, based on experience with some special purpose AI languages in the early 1970's, MICROPLANNER in particular, based on similar ideas. As a result, PROLOG was largely ignored here until the Japanese announced that it would play a central role in their Fifth Generation effort. (However, LISP was largely ignored in Europe during this period, because it was judged too costly in machine resources consumed, and PROLOG was widely experimented with.) After the announcement, it was discovered that some significant breakthroughs—Warren's approach to compilation being particular noteworthy—combined with PROLOG's willingness to sacrifice logical purity for efficiency, had resulted in a programming language very well suited to some AI problems. In fact, it turns out that many of the programming techniques listed above can be implemented in PROLOG in a perfectly straightforward fashion [7].

Both LISP and PROLOG have meta-level introspection facilities that support a deeper level of *computational introspection and reflection* [39] than conventional languages, such as C and Ada. This fact has some impact on the scope and limits of certain V&V approaches; see Section 8.

3.2.2 Higher level tools

Higher level tools can be (roughly) divided into high-level programming environments and expert system shells. A *shell* is typically the result of abstraction from an existing knowledge-based system. It consists of an inference engine, an empty knowledge base that is populated for the particular application, and some support tools. A *high-level environment* is a collection of tools—inference engines, knowledge representation languages, and so on—that have been at least partially integrated, so that the user can choose among the various options. EMYCIN is the classic example of a shell, but more are constantly becoming available, often specialized to a particular domain so as to supply even greater leverage. A good example is Verity's TOPIC system for text retrieval: the developer provides a definition of the concepts to be used in retrieval and the text database, and TOPIC provides everything else. S.1, KEE, ART, PROKAPPA, and so on, are all representative examples of high-level environments.

But for both shells and high-level environments, the impact of the tools on choice of V&V methods depends only on those programming techniques made available by the tool that are used by the developer. No direct matching of methods to tools is necessary (or, in the case of environments, desirable). This is fortunate, as the most popular tools at the time you are reading this manual are likely to be different from the ones available when it was written.

3.3 A Categorization of Knowledge-Based System Architectures

Categorization of AI architectures in general is a difficult task, which will not be attempted in this manual. Even paradigmatic AI systems can employ ad hoc architectures that seemingly cannot be described in any way that provides guidance in defining a verification and validation methodology. However, categorization of a restricted--but important!—subclass of AI architectures, those employed in knowledge-based systems, is feasible. For our purposes, a knowledge-based system is one that can be divided into a knowledge base, an inference engine that derives conclusions (perhaps together with explanations) from the knowledge base, and support software (such as the user interface). The knowledge base might consist of logical assertions, condition-action rules (perhaps with associated confidences), a Bayesian network, or any other natural representation of domain knowledge. The inference engine can be arbitrarily complicated, from a simple algorithmic scheme such as backward chaining from a goal or propagation of confidences in a Bayesian or quasi-Bayesian fashion, to a knowledge-based system in which a "meta-level" inference engine determines how to control use of the "object-level" domain knowledge based on the contents of a separate "meta-level" knowledge base that contains an explicit representation of control knowledge. Much of the second-generation knowledge-based expert system work in the early-to-mid 1980's was driven by the recognition that it was useful to factor out an explicit representation of the control knowledge, because doing so

- increases modularity, which makes the system easier to develop and modify [16, 14, 1],
- makes the domain knowledge more reusable [14],
- facilitates explanation generation [46], and
- places the focus on "representational adequacy" rather than on efficiency in representing domain knowledge [19].

Verification and validation of simple inference engines is straightforward. It should be proved that the inference algorithm produces the desired conclusions. That the code correctly implements the algorithm should be verified using conventional verification techniques. The system should be validated by exercising it on representative and extremal knowledge sets. That is, a simple inference engine is verified and validated as if it were "just another algorithm." There is no special "AI V&V" problem involved. Therefore, the taxonomy of knowledge-based systems is heavily slanted towards more complex inference procedures, where special problems arise. The categorization, adapted from the one in [19], is based on how much effort is put into determining the next inference step or series of inference steps, and on when this "meta-level" inference is performed. This categorization ignores factors—such as, whether the inference is performed at assertion-time or at inference-time—that are essential to design of the system, but are largely irrelevant to how V&V should be performed.

Meta-level inference architectures are distinguished by the fact that, at any given time, the system can be active at either of two levels. It can be determining *what to do*, by drawing conclusions from the meta-level control knowledge, or it can be *doing it*, by drawing conclusions from the object-level domain knowledge. Thus, there is a spectrum of "how meta" a meta-level inference architecture is, depending on how much time it spends working at the meta-level, from systems that perform very little meta-level inference to systems that perform meta-level inference almost exclusively. A system that spends very little time working at the meta-level will be called a *object-level-oriented inference architecture*. A system that expends the bulk of its effort at the meta-level will be called a *meta-level-oriented inference architecture*. A system that puts substantial time and effort in at both levels will be called a *mixed-level inference architecture*.

Mixed-level architectures will be further subdivided according to the conditions under which resources are devoted at the meta-level. Most common are *reflect-and-act* systems, which perform meta-level reasoning before (or, equivalently, after) each object-level step. Blackboard systems commonly use a reflect-and-act loop to apply control knowledge in a particular situation. If meta-level reasoning is used only when some crisis develops at the object-level—where a crisis can be anything from having too many options available to recognizing an inconsistency in the object-level knowledge base—we have a *crisis management* system. Finally, if the primary function at the

meta-level is to divide inference tasks into subtasks, which are then handled at the object-level, we have a *subtask management* system. This completes our taxonomy, which is graphically represented in Fig. 3-2: the influence of particular architecture categories on choice of V&V techniques is discussed in Section 7.

3.4 Common Characteristics of Typical AI Problems

As was mentioned above, the nature of typical problems addressed by AI has some impact on how to perform V&V of AI systems. There are two principal relevant characteristics.

1. The system requirements are hard to define. This is partly because of the size of the problems addressed: AI is often applied to problems too large to deal with by more conventional algorithmic techniques guaranteed to produce an optimal solution. It is partly due to the intrinsic nature of the problems: AI is often applied when the problem is too vague to be addressed by techniques that operate on precise measurements. In either case, it may be impossible to define what counts as a solution to the problem being addressed, *even after the system has been completed.* There just are no cut-and-dried criteria that can be applied to determine whether the system "does the job."

2. The solution process is knowledge intensive. Sometimes computers are used to solve problems involving tedious and repetitive, but ultimately simple, calculations of some sort. No real understanding of the problem domain is needed: a human could solve the problem without knowing where the numbers came from or what they represent. Solving an AI problem more often requires a detailed formal model of the domain that captures complex interdependencies among various domain elements. This knowledge can be hard to break up into easily digestible pieces, which makes many of the programmer's tools for managing complexity inapplicable. A complicated domain model can be harder to debug than the same number of lines of "spaghetti code."

Both these characteristics are illustrated by considering some typical AI problem domains:

- planning problems too large to handle with the methods of Operations Research, involving many soft, evolving constraints and imperfect knowledge of the conditions under which the plan will be executed,
- image and signal understanding problems that require discovery and analysis of subtle patterns in reams of data,
- natural language understanding (no more need be said about this one!).

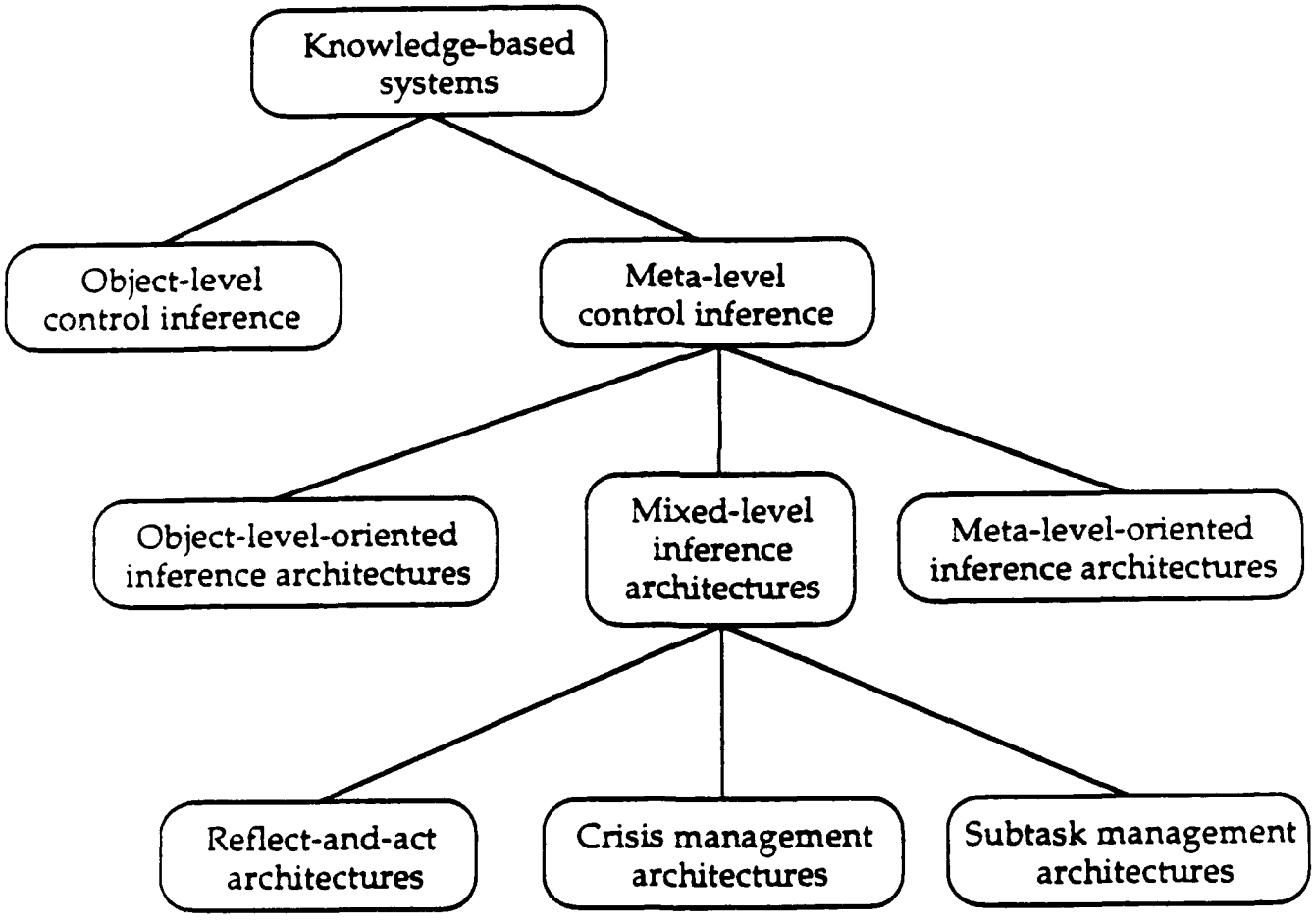

Figure 3-2: Knowledge-Based Systems Taxonomy

- medical diagnosis (ditto!),

and so on.

The impact of these characteristics on V&V will be considered in Section 4.

3.5 Common Characteristics of the AI Software Lifecycle

The most important characteristic of the AI lifecycle is that systems typically evolve through refinement of a prototype that is used to help determine system requirements. Even in cases where some of the code can be developed in accordance with some sort of waterfall model, other parts are constantly evolving. Thus, even more than in conventional systems, there must be an emphasis on reverification and revalidation of modified systems. The impact of this emphasis on choice of V&V methods will be discussed in greater detail in Section 4.

4. Development Models and Methods

Is defining a single, uniform standard AI development methodology desirable? AI development efforts vary too much to impose strong methodological restrictions, and weak methodological restrictions are useless. However, that does not mean that no guidance can be provided. That guidance will be in the form of advice on how to determine an appropriate methodology for a project, rather than methodological rules that should always be followed. Here is an example of such a "metamethodological rule," one that we urge you to adopt.

> *One of the first tasks on every project should be to define the system development methodology that will be employed.*

Defining a methodology requires choosing a software development model. If the software development process needs to be tailored to the needs of each AI application, then providing a meta-level model that describes how to define a software development process appropriate for particular AI project will prove more useful than "yet another software development model." The four dimensional model described below represents an attempt to provide such a meta-level model.

4.1 Software Life Cycle Models

The traditional software life cycle model developed initially in [36] is a one dimensional model that portrays all software development projects as passing through the same linear progression of phases. Over the years there have been many different elaborations of this linear model—each is usually proposed as universally applicable to all software. More recently, a two dimensional spiral development methodology has been proposed by Boehm [6] and is gaining widespread acceptance because it explicitly recognizes that rapid prototyping is often useful before attempting to define the requirements for the proposed system.

In this report we propose a meta-level model of alternative software development processes. This meta-level model is especially appropriate for AI software where there are strong concerns about verification and validation. In this generalized model, which is then specialized to fit the needs of any specific software project, software products (such as requirements specifications, performance prototypes, or a regression test) are represented in a four dimensional space. The advantages of this model are:

1. Rather than providing a single model as a template for all software projects, our meta-level model describes how to define a development process tailored to the specific needs of an application—one that still meets V&V needs. Any software development model needs to be adapted to fit the needs of a specific project, but most give little or no information about what adaptations are valid.

2. It focuses on the products of software development more than on the development process. Across different software applications there is more commonality in the final products than there is in the sequencing used during the development process.

Our meta-level model encourages the definition of specific software development plans that produce not only the products defined by existing software development standards such as DOD-STD-2167A but also the additional products associated with rapid prototyping and/or formal specifications. DOD-STD-2167A explicitly allows rapid prototyping methods to be used during software development but does not provide guidelines or a framework for when and how to use rapid prototyping and what should result from different forms of rapid prototyping efforts.

Both the conventional software development methodology and the spiral model are common special cases that can be constructed using our meta-level model. Since many AI systems are embedded in applications with conventional software, it is important that the model for developing and validating AI software be a generalization of conventional development methods.

Our general model recognizes that it is often best to prototype the hardest aspects of the problem—the prototype may deal with functionality, with the user interface, or with the execution time of key functions. In other applications, formal specifications or other intermediate software products are useful as developmental waypoints while working toward the final software products.

DOD-STD-2167A specifies that a software project should develop and use its own individualized Software Development Plan and a Software Test Plan. Our software development model is intended to provide a framework and guidelines that AI software developers can use to develop these plans in a way that is tailored to the needs of the project and is compatible with AI development methods being used. The parts of the test plan that deal with verifying the software can be tailored both to the specific needs of the project and to the intermediate software products that are appropriate for this project.

The problem with a software development model that is uniform for all software projects is that either it constrains the development plan to fit into the constraints of a single, pre-conceived developmental process or else it doesn't model the software development process in very much detail. For example, it is widely recognized that the waterfall model should be adapted to the needs of an individual project. But the waterfall model does not go on to provide a framework for that adaptation. Our meta-level model provides a framework for constructing more detailed development models and V&V plans that are tailored to the specific needs of the application.

4.2 The Four Dimensional Software Development Model

Software development begins from a vague, partial, and ambiguous understanding of the problem to be solved, and it progresses toward a set of products that define an unambiguous, executable, and efficient representation of a problem solution. The evolution, verification, and validation of these software products can be represented by paths through a four dimensional space where the dimensions are labeled and scaled as follows:

Definition:	Partial understanding of problem	→	Complete understanding of problem
Formalization:	Ambiguous	→	Unambiguous
Operationality:	Abstract	→	Executable
Efficiency:	Inefficient	→	Efficient

These four dimensions correspond closely with what Hoare characterizes as the four components of a complete theory of programming.

> A complete theory of programming includes
>
> 1. A method for specification of programs which permits individual requirements to be clearly stated and combined.
> 2. A *method for reasoning* about specifications, which aids in elucidation and evaluation of alternative designs.
> 3. A method of developing programs together with a proof that they meet their specification.
> 4. A method of transforming programs to achieve high efficiency on the machines available for their execution.
>
> *(From Hoare's "Foreword" in [12], p. iii.)*

There are strong interactions between these four dimensions, so it is often difficult to address them individually; however, software complexity can be reduced and verification and validation can be more effective when there are software products that address each of these four dimensions separately.

Logically, it may seem possible for software development to proceed sequentially through each of these four dimensions. Unfortunately, there are strong backward dependencies among these dimensions; that is, progress in one dimension often depends on prior progress in a later dimension. For example, before defining all the requirements for a software system, one may need a prototype implementation in order to understand what is practical and feasible as requirements. Similarly, with current software techniques, it is seldom practical to defer performance considerations until after a formal specification and an initial operational program have been completed.

These backward dependencies among the four dimensions are the primary reasons why each software project needs to have its own Software Development Plan. These backward dependencies are different for different software applications and development environments. A Software Development Plan should address these backward dependencies by working out the relative order in which each component of the system will make progress in each dimension. Our four dimensional, meta-level model applies both to the software system as a whole, to each of its components, to their components, etc. The effort to divide a software system into relatively independent components is one of the main tasks of software design, and it is not dealt with by the four dimensional model. This model focuses on the different kinds of information to be recorded about either the whole software or about any component. During development, it should be expected that different components of the overall software will have been developed to different stages at any one time. A detailed development plan for a software project will identify the software components and the relative order in which each component will progress through the four dimensions.

The four dimensions are discussed in the following paragraphs.

Definition. This dimension is the focus of a requirements specification. When a software project is originally conceived, one has only a vague understanding of the problems to be solved by automated processing and of the functions needed to solve them. The definition dimension measures the degree to which the functional and performance requirements of the system have been understood and defined in some form.

Formalization. This dimension measures the degree to which the software is defined in some formal, precise, and unambiguous notation that supports deductive reasoning. Formalization removes the ambiguity that may be present in a requirements specification. More important, formalization facilitates deduction of properties about the system; thus enabling properties that are implied by a specification to be made explicit.

Operationality. A system can be defined (informally or formally) at an abstract level without mapping this definition into constructs that are executable on available computer hardware. This dimension measures the degree to which the system functionality is not only defined but also executable. Progress in this dimension can be thought of as mapping a system down to lower levels of abstraction—from application-oriented concepts through computer abstractions like queues, stacks, and processes and on down to bits and bytes.

Efficiency. This dimension deals with all of the characteristics of the system that are separate from the input-output functionality: execution time, response time, memory and other resource utilization, etc. Programmers usually deal with both operationality and efficiency concerns together during the implementation process; however, operationality and efficiency are sometimes separable concerns.

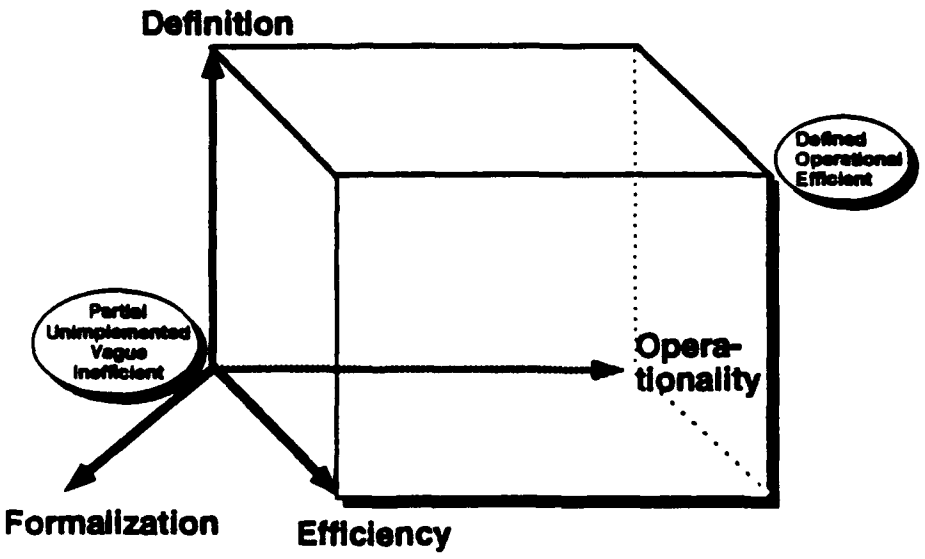

Figure 4-1: The Four Dimensional Space for Software Development

For example, functional prototypes can be distinguished from performance prototypes.

4.3 Evolving Software through the Four Dimensions

Unlike the waterfall models of software development, this four dimensional representation does not itself describe stages in a software development; rather the product of a development stage is represented by a point within this four dimensional space. (More precisely, a development stage usually results in several products that are represented by a set of points together with mappings between those points.)

Figure 4-1 shows a representation of the four dimensional space for software development. The definition dimension is along the vertical axis, operationality along the horizontal axis, and efficiency is represented on the axis coming forward. The other axis going to the lower left in the fourth dimension represents the formalization dimension. Since it is difficult to visualize a four dimensional space, our figures usually will not represent this dimension.

The need for documentation to justify that the program meets its goals and to support maintenance and future evolution can be satisfied by products represented by points in the four dimensional space or by mappings between these diverse products. A minimal set of software products as specified by DOD-STD-2167A would be a Sys-

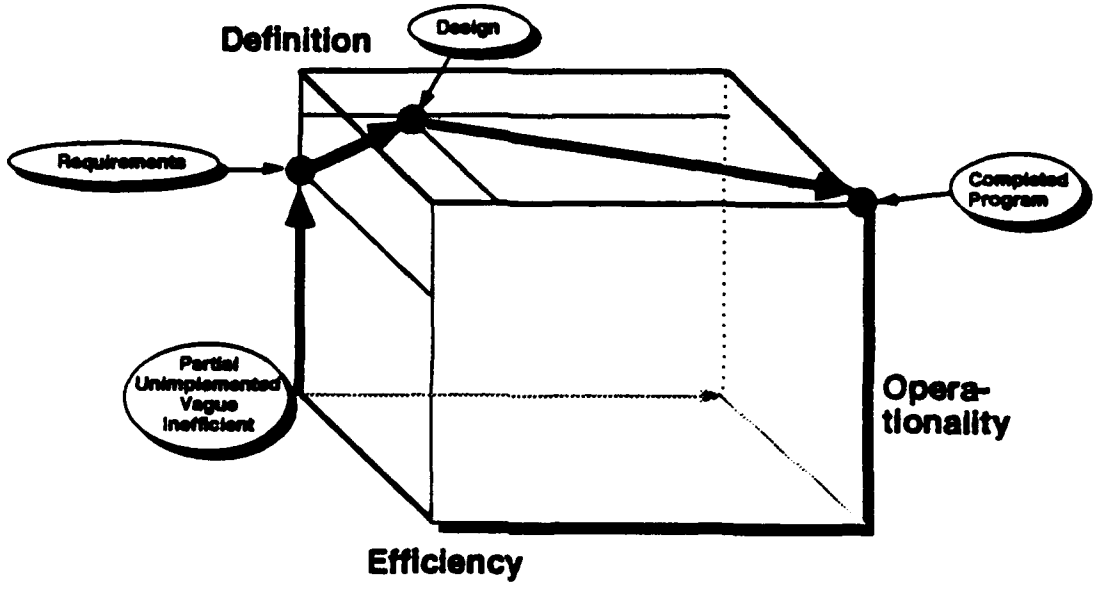

Figure 4-2: The Minimal Products of Software Development

tem Requirements Specification, a Software Design Document, the Source Code and Listings of the completed program and the Software Test Reports.

These products are represented in the four dimensional space as shown in Figure 4-2. The Software Test Reports are represented not by a point in the space but by the mapping from the completed program to the requirements. (Other documents required by 2167A are a Software Development Plan and a Software Test Plan. The Software Development Plan defines a path that the project will follow through the four dimensional space. The four dimensional representation is an aid for designing the Software Test Plan, but the plan is not represented in the four dimensional space. The Operator's and User's Manual are also required products but are orthogonal to the issues described by the four dimensional model.)

Our generalized model allows a wide variety of other possible intermediate software products to be defined as points in the four dimensional space. These additional software products can be used as additional waypoints during software development; thus the standard products required by 2167A are particular cases of the software products that can be represented using the four dimensional model.

Waterfall models specify a fixed sequential process for generating these software products. In our four dimensional model, we represent the software products that result from a waterfall model, but we do not define a fixed, sequential development process; and we make it easier to include additional products as waypoints during software development.

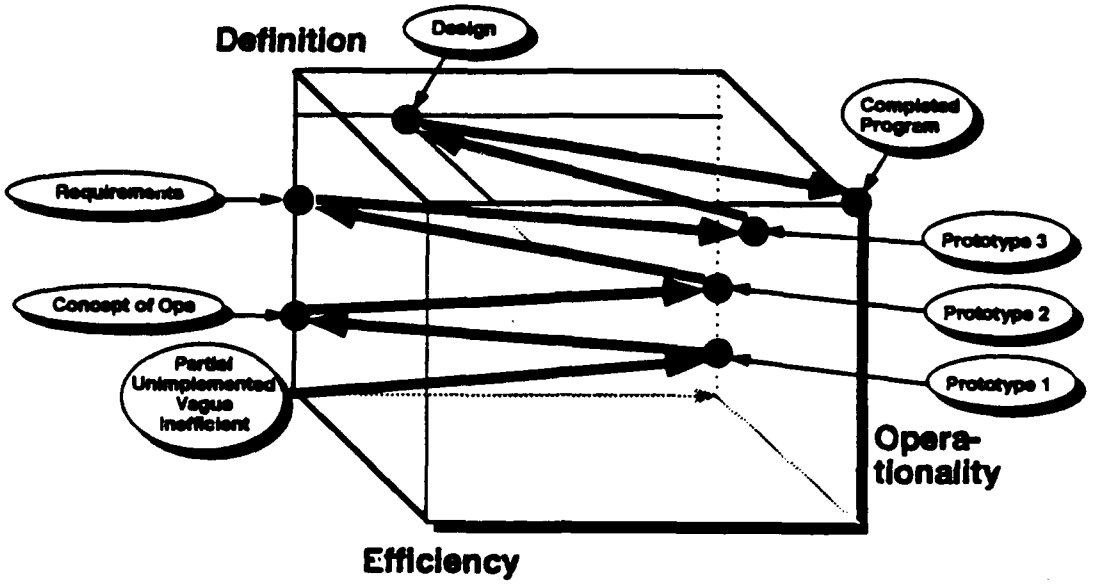

Figure 4-3: The Spiral Development Model Represented within the Four Dimensional Space

4.4 Rapid Prototyping and the Spiral Development Model

Boehm [6] describes the software development process as a spiral. Each cycle of the spiral is devoted to resolving the highest risk issues involved in this particular application. Experience from the implementation and use of each prototype is used to develop a more complete definition of the functionality and performance that is desired in the final product. Figure 4-3 shows how the products of a typical spiral development would be represented within the four dimensional model. One begins by prototyping and defining the concept of operation for the application. Then one prototypes and defines the software requirements, and finally one builds a structural prototype leading to the definition of the software design. The completed program for the system that is to become operational is then implemented based on this design. An advantage of the four dimensional model is that it provides a framework for thinking about alternative development processes that use alternative waypoints in order to arrive at and verify the final products of the software development process. Requirements documents, design documents, and various forms of prototypes can all be understood as intermediate software products associated with particular points in the four dimensional space.

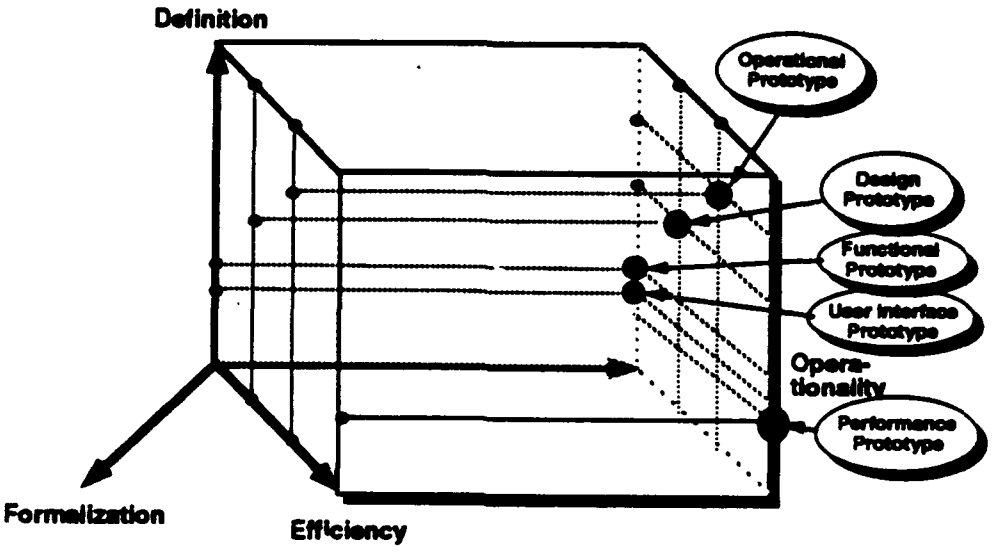

Figure 4-4: Different Kinds of Prototype Implementations

With respect to prototypes, it is important to specify what functionality or efficiency characteristics one is trying to prototype. The implementation effort involved in building a prototype will be large if one does not carefully focus the prototyping activity on critical issues. Figure 4-4 shows five different kinds of prototypes located at different points in the four dimensional space.

Functional Prototype. This is an implementation of some critical functionality to test whether it can be implemented successfully. For example, if a large system is being built that includes automatic recognition of targets from certain kinds of images or signals and if similar functionality has not been demonstrated in previous systems, then a functional prototype to test the feasibility of achieving this functionality is indicated.

User Interface Prototype. If the interaction between the system and the user is to be quite different from that of any previous system, then a prototype that does not have much internal functionality but simulates the proposed user interface and evaluates it in actual tests with representative users is indicated.

Performance Prototype. If some of the functions required in a proposed system have never been implemented with adequate response times using the proposed processing resources, then a prototype to test these functions running on the proposed hardware (or a simulation of it) is indicated.

Design Prototype. A design or structural prototype is used to evaluate the effectiveness of a proposed software design. It typically addresses a combination of critical functional and performance issues.

Operational Prototype. Sometimes it is useful to test the majority of the functionality of the proposed system running in an environment similar to the ultimate operational setting. As represented in the four dimensional space, an operational prototype may involve much of the expense of a completed product, and it is usually useful to plan it so that the majority of it can be evolved for use in the final product.

One advantage of the four dimensional model is that the differences between different kinds of prototypes are explicitly represented in the model.

4.5 Formal Specifications as Software Development Waypoints

Different kinds of prototypes are not the only additional software development waypoints that are represented with the four dimensional model. By considering the dimension dealing with formalization, different forms of specification can also be associated with points in the four dimensional space. These various specification options are important for V&V. Figure 4-5 shows some of these specification options as points in the four dimensional space.

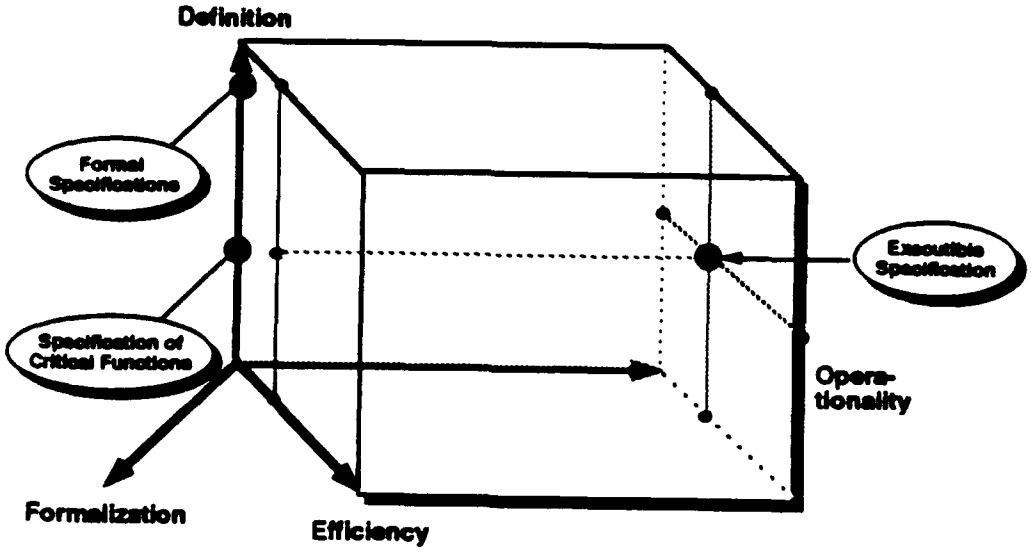

Figure 4-5: Additional Possible Specification Waypoints

4.6 V&V as Mappings between Software Products

In our meta-level software development model, verification and validation are not represented as points in the four dimensional space; rather, they are represented as mappings between points. Both verification and validation are concerned with showing that the implemented software satisfies the software requirements. Validation does this directly by testing and other methods that directly evaluate the software implementation against requirements. In Figure 4-6, validation is shown as a direct mapping between the implemented code and the software requirements document.

Verification uses an indirect path through intermediate software products to show that the final software meets the requirements. The indirect path that is used in conventional software development is shown in Figure 4-7. We propose that the specific choice of the intermediate waypoints for verification are less important than the basic process of showing that the final executable software maps through some series of intermediate waypoints back to the system requirements.

The reason for doing verification using intermediate waypoints is that the mapping from one waypoint to the next can be much simpler than the mapping all the way from the final software back to the requirements; and additional techniques can be applied to verify that the software product at one waypoint satisfies the requirements imposed by the previous waypoint when the conceptual gap between the two waypoints is relatively small.

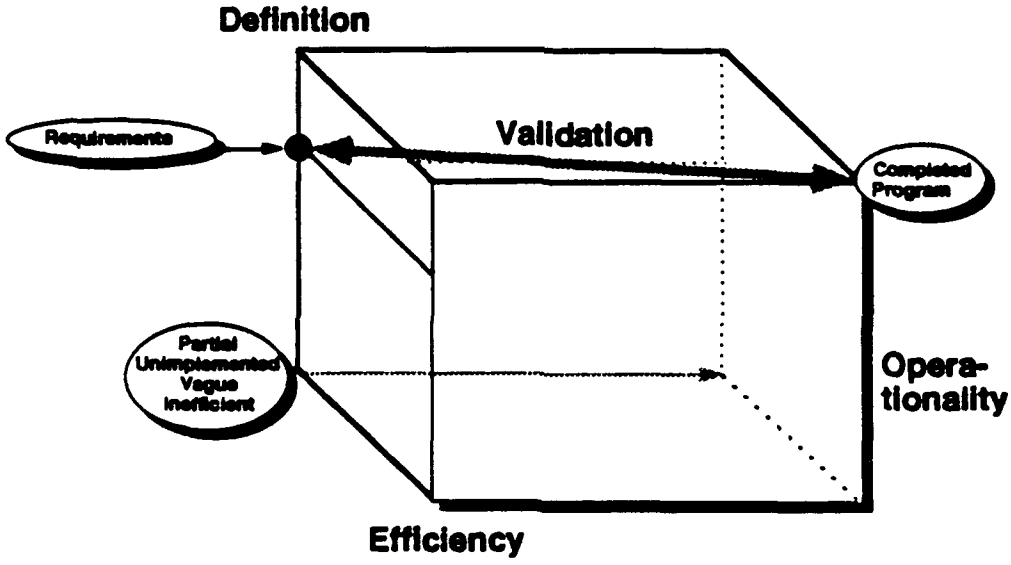

Figure 4-6: Validation as a Mapping between the Implementation and Requirements

Sometimes software verification is viewed as showing that a correct software development process has been followed. We prefer, however, to emphasize verification as focused on showing that the content of the products of each stage of software development satisfy the requirements for those products.

Once one introduces prototype development as a legitimate part of the software development process, one does not want to require that the waypoints used to verify software have to correspond completely with all of the developmental waypoints used during the evolution of the software. A prototype may be used largely in order to define an appropriate requirements specification. Thus various prototype software products may be part of the software development history but they need not be used as intermediate waypoints in the verification plan.

The reason why software verification is needed in addition to software validation is that AI software implementations are so complex that testing the software directly against requirements cannot be complete. Software complexity also makes verification difficult; however, there are techniques for reducing the complexity of the mappings from implemented software through intermediate waypoints to a requirements specification. As discussed in [27, 29], examples of these techniques include:

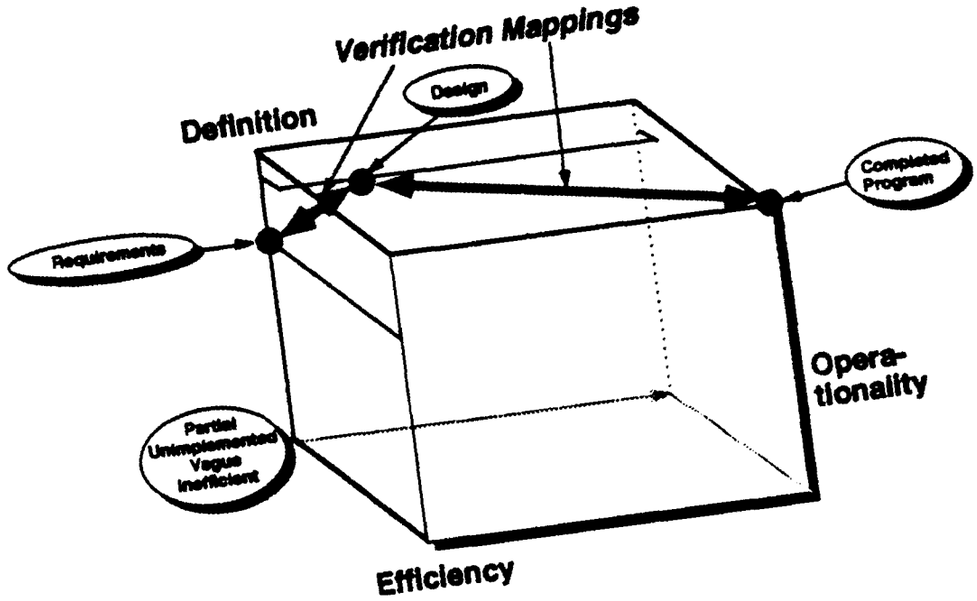

Figure 4-7: Verification as a Sequence of Mappings between the Implementation and Requirements

- Automate parts of the evolution from the requirements specification to the implementation. Compilers for high level programming languages automate part of the work in the direction of operationality, and automatic programming techniques that are capable of executing and optimizing specifications written in very high level languages can contribute further automation. It is still necessary to verify that the automatic compilation preserves correctness but automation means that this verification effort need not be done repeatedly.
- Use software engineering techniques to constrain the design and the implementation so that the mapping back to the requirements specification is more understandable and less complex.
- Focus on verifying the most critical functionality in the requirement specification.
- Make the satisfaction of critical requirements depend on only small parts of the implementation.
- Separate control and efficiency issues from the functionality of the implementation.
- Make the implementation be partially self-checking.

4.7 *Work Remaining on the Meta-level Development Model*

More work remains to elaborate this four-dimensional software development model before it will be useful as a guideline for developing and verifying AI software. In particular, we need to develop and test:

- Metrics for measuring progress in each of the four dimensions.
- Definitions for additional waypoints that are useful during software development and verification.
- Examples of how different sets of waypoints can be used successfully in different kinds of AI software development projects.
- Additional methods for doing verification by mapping between these waypoints.

Once these waypoints and metrics are defined, a software development and verification plan can be formulated as a set of waypoints or stepping stones that are located within this four dimensional space. Planning a software development process is then analogous to identifying stepping stones for crossing a river. One tries to find a series of stepping stones that are close enough so that software development can proceed in a series of relatively small conceptual steps with each step being reliable

and verifiable. One does not always have to use the same stepping stones, one just has to find stepping stones that are close enough together so the complexity of the conceptual gap between them allows verification techniques to be applied successfully. If we think of the complexity of the complete software system as analogous to the width of the river, then this analogy indicates that the number of waypoints that are needed in a software development plan is not the same for all software projects but is proportionate to the complexity of the software.

4.8 Conclusions about the Meta-level Software Development Model

Each software project needs to define the specific software products that are to be produced by the project and the order in which they should be generated. Our four dimensional meta-level model provides a way to represent both the final products, the intermediate products that are used as way-points in working toward the final products, and the validation and verification mappings between them.

The four dimensional model is useful in laying out a Software Development Plan that is tailored to the needs of the specific project. Certain software products will appear in the plan for every project—for example, requirements specifications and design specifications. The level of detail that is achievable in these products may, however, differ with different projects. Prototypes and formal specifications, however, need to be planned to meet specific needs of each project. The verification and validation plan may also vary with specific project needs. Prototypes that are developed primarily to understand the application requirements will probably not play a major role in the V&V plan; however, an executable specification may serve as a prototype and also be used as an intermediate waypoint in verification by showing that the final optimized code is equivalent in functionality to the executable specification which itself satisfies the requirements. We believe that this four dimensional model for software development will provide the flexibility and adaptability needed to tailor AI software development to the needs of the application while also providing a framework with enough structure so that verification, validation, and maintenance of the software can be effective.

Part II

HOW TO SUPPORT V&V

5. For Technical Leaders: Planning to Support V&V

This section is devoted to providing technical leaders on AI projects with some guidance on how to support V&V during software project planning. It touches on a number of issues that are addressed more fully in subsequent sections, as well as on issues specific to planning.

5.1 What Level of V&V is Appropriate?

Perhaps the most fundamental example of a planning-specific V&V issue is how to determine the *level* of V&V that is appropriate for the project. Better verification and more comprehensive validation can always be obtained by devoting more resources to the V&V process. But resources are limited, and some of those resources must be devoted to other tasks. Before the detailed tradeoffs among potential V&V procedures are determined, the total level of effort that will be devoted to V&V should be settled. While both the level of V&V and the particular procedures to be applied are often subject to external constraints—organizational policies or contractual requirements, for example—that somewhat simplify the problem, a careful analysis of the "natural" V&V requirements on the project is still essential to making the best use of scarce development resources.

Guideline 5.1 *Determine the total level of effort to be devoted to V&V, based on the characteristics of the project, at the beginning.*

5.2 Project Categorization: An Example

One generally useful technique for determining the level of V&V for a project is to use a categorization, based on salient characteristics of the project, to determine what sorts of V&V activities are necessary and appropriate. The details of such a categorization should be determined by the sorts of projects that your organization performs. Below you will find an example, based on a categorization found useful at the organization where the authors are employed, that should provide a useful baseline for your category definition efforts.

First, here are the criteria used in determining the category of a project. Each criterion is rated as being satisfied at a Low level, a Medium level, or a High level on the given project. Table 5-1 explains what the three ratings mean for each criterion. (Note that some of the criteria are marked with an asterisk, '*'. The significance of this will be explained in the definition of the categories.)

Our three categories of projects are simply called A, B, and C. Which category a project belongs to is determined from its ratings, as follows.

	Criterion	Rating=Low	Rating=Medium	Rating=High
*	User of system	Developer only	Trained Computer Scientists	Average folks
	Contract Type	CPFF, T&M	FPLOE, CPAF, CPIF	FPP (Completion)
*	Flexibility in Deadline for Delivery	No requirement	Negotiable	Fixed and non-negotiable
	Delivery Environment (H/W and O/S)	Exotic (e.g., Connection Machine)	Unix workstation	PC microprocessor board
	Coupling to Existing Systems	Standalone workstation	Loosely coupled	Embedded
*	Execution Speed Requirements	Easily met	Difficult to achieve speed perceived as satisfactory	Hard real time requirements must be met
	Design Formalism	Developer's choice, designs not subject to customer review	Designs subject to review, familiar design formalism	Customer-defined unfamiliar design formalism
	Database/Knowledge Base	DBMS/KBMS of developer's choice, data already recorded, knowledge already formalized	Developer's choice, but substantial data acquisition/ knowledge engineering effort required	Must interface to COTS software of customer's choice, substantial effort devoted to populating DB/KB
	Documentation	Simple User's Manual only	User's Manual, Maintenance Manual	Full 2167A
*	Testing Requirements	Demo only	Testing at system level must be documented	Unit testing must be documented
	Configuration Management Requirements	None	Manual CM	Automated CM tool employed
	Support Software to be Developed	None	On-line help system	On-line help, tutorial, system status monitoring, etc.

Table 5-1: Criteria Used in Categorizing Projects

- If a project rates a High on any of the asterisk-marked criteria, its category is A.
- If a project rates a maximum of Medium on the asterisk-marked criteria, then
  - if it rates a High on the majority of the unmarked criteria, its category is A,
  - else its category is B
- If a project rates a Low on all asterisk-marked categories, then
  - if it rates at least Medium on the majority of the unmarked criteria, its category is B,
  - else its category is C.

Each category has an associated baseline level of documentation that must be produced and procedures that must be followed to support V&V on the project. (See Appendix B for definitions of acronyms.)

A. Category A projects must
   - produce a Software Development Plan prior to development,
   - produce an SRS, IRS, SDD, IDD, STP, STD, and STR during development,
   - define and follow formal configuration management procedures,
   - appoint a qualified System Engineer as Program Technical Director,
   - institute a formal Quality Assurance program,
   - perform formal testing,
   - employ a formal design method,
   - publish a detailed schedule,
   - institute a process for collecting and handling SER's,
   - conduct formal Engineering Reviews (design reviews, code walkthroughs, etc.), and
   - participate in the company's TQM program by conducting quarterly Program Reviews.

B. Category B projects must
   - produce a Software Development Plan prior to development,
   - produce an SRS, SDD, and STP during development,
   - define and follow configuration management procedures,
   - appoint a Program Technical Director,
   - publish a schedule,

- conduct (possibly informal) Engineering Reviews, and
- participate in the company's TQM program by conducting quarterly Program Reviews.

C. Category C projects must

- define high-level goals and plans for the project,
- publish a schedule of major milestones, and
- participate in the company's TQM program by conducting quarterly Program Reviews.

This sort of categorization provides the level of V&V appropriate to the project, and allows at least a rough prediction of the percentage of project resources that must be devoted to V&V, while leaving the procedural details open.

6. For System Specifiers: What and How to Specify

6.1 The Role of Requirements Analysis in AI Development

Requirements are properties a system must have in order to solve the problem. The bulk of a requirements definition document will often be a statement of the problem. Indeed, the most difficult part of requirements analysis is typically achieving a sufficiently detailed understanding of the problem. However, leaving the actual statement of requirements at the level of "the only requirement is to solve the problem" is usually unsatisfactory because of vagueness in the problem statement. Requirements should be defined sufficiently precisely that a given system either satisfies them or does not satisfy them. Moreover, it must be possible to determine—with a degree of certainty that is itself a requirement on the system—whether a system satisfies the requirements or not. (Certification of requirement satisfaction is accomplished via a combination of verification—formal, informal, or some combination of the two—and validation testing.)

Guideline 6.1 *System requirements should be spelled out in enough detail that a system either satisfies them or fails to satisfy them: do not leave the requirements statement "fuzzy."*

As was pointed out in Section 2, it is important to remember that *the product of requirements analysis is not necessarily the true requirements.* Use of the term "requirements definition" is appropriate from the contractual point of view—where a system will be accepted if and only if it satisfies the requirements *as defined in the contract*—but somewhat misleading. It is quite possible that a system satisfies the requirements as stated yet fails to solve the problem, due to an error in the requirements analysis or a loophole in the requirements definition. Systems validation should be performed against the true requirements; satisfaction of the stated requirements is evidence that the requirements are satisfied, but it is not conclusive evidence. The important point to keep in mind is that comprehensive testing against the stated requirements cannot, in general, provide comprehensive validation. A requirements definition document is just another lifecycle product, and no amount of verification that the system satisfies the stated requirements can eliminate the need for validation testing.

Guideline 6.2 *Never confuse the real requirements with the stated requirements, because there is no guarantee that the requirements have been stated correctly.*

A common characteristic of AI system development efforts, as noted in Section 4, is that the requirements are vague and hard to define. This is partly due to the nature of the problems addressed by AI. Just as it is hard to define a performance measure that

captures how well an expert does his job, it is hard to define a performance measure that captures how well an expert system performs the same job. It is also partly due to the fact that AI techniques were developed to address large, comparatively ill-defined problems. A good example is search. Conventional techniques perform well when the search space is comparatively small, allowing effectively exhaustive exploration of all alternatives, and there are well defined criteria for what counts as a good solution. When AI is applied to search, it is typically because the search space is too large or too ill-behaved for conventional techniques, or because there are no effective criteria that distinguish good solutions from others. Often, the best you can do in such cases is to build a system that emulates a person who is believed to perform his job effectively. This is just the sort of case where there seems to be little hope of accurately defining the requirements.

So, does this mean that AI development projects should not waste effort on attempting to define system requirements? The typical AI lifecycle, based on iterative refinement of a prototype, might seem to support an affirmative answer. The developer asks the user whether the system is satisfactorily in some given respect; if the user is not satisfied, he is asked how the system must be changed so that he will be. The end product of this refinement process is the prototype that embodies the requirements, rather than a document that attempts to state the requirements. What, if anything, is wrong with this? Doesn't the prototype provide even better guidance in the subsequent evolution of that system than written requirements would?

The answer to that last question is a resounding "No!" To see why, we must take a closer look at the role of requirements statements in the system lifecycle. Requirements play a quite different role in the lifecycle than specifications and designs. Requirements are determined by the problem itself, and cannot be changed by the implementors. If a system does not satisfy the requirements, it does not solve the problem (although it may solve some closely related problem). Thus the requirements explicitly present the constraints on the design process. And this is the key to seeing why an explicit requirements definition is essential.

Consider, for concreteness, a mock-up of a system's user interface. A user-approved mock-up of the user interface cannot usually be considered an adequate requirements definition for the interface, for several reasons. First, some features of the mock-up that were considered artifacts[1] by its creators may have been essential in securing the user's approval. Perhaps color of some green icon was selected at random by the developer, yet the user would not have approved the interface if the icon had been any other color. Unless the *essential* features of the user interface are distinguished from the *accidental* features, people who subsequently want to change the interface, for one reason or another, have no guidance on what can be changed and what must

[1] "Artifact," as it is being used here, is a technical term. It means that the feature is the result of an arbitrary choice made by the developer and that it is not intended to be a feature that the actual user interface must share in order to be considered essentially the same as the mock-up.

remain the same. Second, "tolerances" are left implicit, so questions such as "Is it acceptable to make this window 10% narrower?" cannot be answered.

Thus, while prototyping is a useful—sometimes virtually essential—technique for determining system requirements, the prototype alone is not an adequate requirements definition. Ideally, the prototype would be supplemented with a list of all its relevant features, classified as essential or accidental, together with a range of acceptable variation in each essential feature. This ideal may not be achievable. It may not even be an appropriate goal on a given project. The point is, a requirements definition is suppose to say what features the system *must* have, and this information cannot be extracted from the prototype alone. Thus a prototype does not provide the same guidance to designers, implementors, and maintainers that a requirements definition does. Far from reducing the value of writing a requirements definition, prototyping *increases* it by helping to ensure that the requirements are correct and sufficiently detailed that they will provide practical guidance!

And the problem becomes even more acute when modifications are made to a system that has already been validated. As a simple example, consider the addition of a new window, which will display a new sort of information that has become available to the user. To what extent can other windows be moved or resized to accommodate this change? The original requirements on their size and position provide an excellent starting point for answering the question: if one window was already as small as requirements allowed, while another was far larger than required, then shrinking the latter rather than the former will almost always make the user happier than the opposite. Without some record of requirements and design decisions that determined window sizing, there is no basis for preferring either alternative over the other. While this example is admittedly trivial—the modifier can ask the user which window should be shrunk, or can even get a mock-up of the new interface approved—in more complicated examples, the guidance provided by the requirements would be even more welcome.

Guideline 6.3 *Prototyping helps to define the requirements, but do not attempt to use a prototype as a requirements definition.*

6.2 The Role of System Specification in AI Development

A specification is a description of the system to be built at a "black box" level of abstraction. This description should be *declarative*, not *procedural*, in that it should say *what* the system does without prescribing *how* the system does it.

Thus, requirements and specification are quite distinct. Requirements deal with properties of systems in general; specifications are descriptions of particular systems (albeit at a very abstract level). Every system that solves the problem must satisfy the requirements, but many incompatible system specifications are possible. The problem completely determines the requirements, but merely constrains the specification. A

requirements document says what the system *must* do: a specification says what the system *shall* do. In fact, QA personnel on large aerospace development efforts check that the right word gets used: statements of requirements must contain the word "must," never the word "shall;" specifications must contain "shall" and never "must." A design is a refinement of a specification; conversely, a specification is a top-level design. The "black box" of the specification is broken up into smaller boxes (more formally, *abstraction units*), those boxes are broken up into smaller boxes still, and so on. Eventually, very simple boxes—i.e., ones that are straightforward to implement—are arrived at. The result of this decomposition is the detailed design. The exact nature of the boxes depends on the design methodology being employed, but, in many cases, they correspond to abstraction mechanisms in the implementation language to be employed. In other words, design is simply identification and specification of parts of the system. Thus a general purpose specification language can be employed throughout the design process, making the design stages appear to be successive refinement steps. Alternatively, different design formalisms can be employed at different levels of abstraction. Unfortunately, the latter seems to be standard aerospace practice—at least to the extent of using natural language or simple diagrams for specification and high level design and some sort of pseudocode for detailed design.

This account is at odds with the view that the dividing line between requirements and specification is not sharp, that requirements simply tend to be more domain-specific and specifications more detailed and complete. There is a grain of truth in this view, in that satisfaction of the specification should imply satisfaction of the requirements. Hence, there is a sense in which all the information in the requirements is captured in the specification. However, the view that a specification is obtained by adding detail to the requirements definition is fundamentally mistaken. It is important to avoid the error of conflating the two because requirements and specifications play very different roles during the maintenance phase of the lifecycle.

Guideline 6.4 *Do not confuse specified properties and requirements. There is nothing wrong with a system that satisfies the requirements but not its specification (though the specification should be eventually modified to fit).*

As was pointed out in the previous section, the role of the requirements definition is to provide constraints on the rest of the development process. This document changes only if an error is discovered in the requirements analysis. A specification is, on the other hand, entirely under the control of the developers. Any change can be made at any point in the lifecycle, provided that the requirements are still satisfied and that the change is propagated through the design for consistency. The purpose of specification and design is to arrive at a program that demonstrably satisfies the stated requirements. In other words, from a V&V perspective, the system specification and design serve primarily to replace a large verification problem—directly verifying that a system that executes the program will satisfy the requirements—by a series of smaller verification problems—verifying that the system specified will satisfy

the requirements, verifying that each refinement of the design preserves satisfaction of the specification, and finally verifying that the code correctly implements the detailed design. Note that this is quite consistent with the more standard system development view that successive refinement of the specification makes it easier to produce code that satisfies the requirements. Stepwise refinement makes development easier precisely because attention can be focussed on the limited number of factors that are relevant to verification of each small step.

As the system changes over time, and the specification and design evolve, having replaced the large verification gap by many small verification gaps greatly simplifies *reverification* of the system. Typically, a small change to the code requires only that a few of the design refinement steps be modified and reverified. If, on the other hand, the code was directly verified against the requirements, a small change to the code can have a large impact on the proof, making reverification nearly as expensive as the initial verification.

However, as has been noted several times above, stating the requirements for the typical AI system is hard. As a result, requirements definitions tend to be incomplete and somewhat vague, which makes a convincing verification of a system specification impossible. In cases where there is such a gap, it may be tempting to ignore system verification, and to concentrate on validation. It might be argued that in any case where testing can provide sufficient evidence that requirements are satisfied, there is no need for a specification or design beyond the source code plus comments. How often have you been told "The code is self explanatory?" The main arguments against this view are purely practical.

First, writing code involves making design decisions, whether those decisions are recorded or not. The code says *how* things are done, but not *why* they are done that way. As a result, when it comes time to modify the code, it is far from clear whether a particular design decision—say, the choice of a particular data structure or algorithm—was dictated by the necessity of satisfying some requirement, or whether it was more or less arbitrarily chosen from among a number of alternatives. Suppose that a stringent performance requirement could be satisfied by replacing one sorting routine by another. Further suppose that the original sort was stable, but that the proposed replacement is not. Can the latter be freely substituted for the former, or is stability a necessary property of the algorithm? If the sorting unit was specified, the question can be answered by a quick examination of the design document; if not, the best you can do is make the substitution, run a few tests, and hope. Thus, virtually any change to the system requires extensive *revalidation.*

Second, unless there is a system design process that guides development, the system is likely to be ill-structured, consisting of patches on patches, a result of a series of what were perceived to be the minimal changes necessary to eradicate bugs. Without a design breakdown, *any* subsequent change to the system requires *complete* retesting. There is no way to guarantee that significant effects of the change are confined to a particular program unit. But a validated design gives the developer the capability to

swap one unit for another, provided that it satisfies the same specification. Thus, a solid system specification and design simplify the revalidation process. AI systems tend to be especially complex, especially liable to modification, and especially difficult to verify and validate. Since specification and design documents are essential to reducing the cost of reverification and revalidation, it is especially important to develop and maintain specifications and designs of AI systems .

Guideline 6.5 *In specification and design, focus on reducing the cost of revalidation and reverification.*

It must be pointed out that only in exceptional cases can validation testing replace verification, anyway. As Dijkstra, among others, has repeatedly emphasized, testing can only show that bugs are present, not that they are absent. Even if confidence in the correctness of the specification is lacking, validation can still increase confidence that the system is robust, reliable, and comparatively free of the typical "mechanical" errors---such as being off-by-one in some indexing—made by programmers who do not have a detailed design to work from.

Assuming that the desirability of specifying the system and its parts has been established, the next questions that must be addressed are: What specification method should be used? How much formality is desirable? What is the added value provided by so-called "executable specifications?"

The advantages of formal specifications of system functionality over informal specifications are substantial. Recall that, from the V&V perspective, specifications are introduced to support validation. Informal specifications, whether presented in natural language or using suggestive diagrams that cannot be assigned any precise meaning, can be very useful in attempting to understand the system. But informal specifications only support informal verification arguments, which are typically lengthier and less convincing. The objective is to guarantee that any system that meets the refined specification will meet the original specification, and imprecision in the original or refined specification makes this exceedingly difficult. Moreover, validation of refinements of formal specifications can be at least partly automated. Most tools based on popular diagrammatic methods, such as Structured Analysis [34], perform at least some consistency checking across levels. Tools that support textual specification languages provide even greater support, based on theorem proving capabilities.

Diagrammatic methods are definitely more popular than textual methods, principally because the average system designer finds them easier to use and the average system implementor finds them easier to understand. However, full specifications of the behavior of complicated systems in diagrams are completely incomprehensible and unmaintainable, often amounting to an attempt to represent every state of the system by a box and every state transition by an arrow. As a result, specification diagrams tend to require abstraction that suppresses detail. This suppression of detail limits the scope of verification. Since developers of AI systems are comfortable with formal

textual representations of symbolic information and techniques for manipulating such representations—that is what AI programs do!—text-based specifications (or hybrid methods that combine diagrams with formal annotations) would seem superior for AI development.

Guideline 6.6 *For AI, text-based specifications are superior to diagram-based specifications.*

Textual formal specification languages can be divided into two types. First, there are *algebraic* specification languages, such as OBJ [18] and ACT [17]. In these languages, one specifies a collection of modules, each of which consists of objects, operations on those objects, and equations those operations satisfy. Second, there are *model-based* languages, such as VDM [5] and Z [40]. In these languages, one specifies modules that are built up from mathematical components—sets, sequences, relations, functions, and so on—using a logic-based language. The essential difference between the two is that the former provides a partial *description* of the unit while the latter provides a *mathematical model* of the unit.

Although there are theoretical reasons for preferring one type of formalization over the other, in practice there seem to be few grounds for choosing between them. Some specifications have been written up both algebraically and set theoretically—e.g., a partial specification of the Unix file system [4, 22]—and it seems that translating between the two approaches is straightforward. The set theoretic approach is more widely used in industry, and the support tools available for the set theoretic approach are definitely more mature. On the other hand, the algebraic approach tends to restrict itself to logical languages for which efficient mechanical theorem proving is possible, and thus offers greater long term potential for automated verification support. If you are unfamiliar with the details of the two approaches, examine a good introductory survey [15, 43] and make your own decision. But be aware that the state-of-the-art is evolving rapidly, so the particular systems described in these surveys may no longer be the best candidates by the time you read this report.

Guideline 6.7 *The state of the art in algebraic and model-based specification is evolving rapidly. Make sure you have considered the best current candidate tools before choosing between the two approaches.*

Some programming languages claim to be *executable specification languages*. The idea is that they provide the capability to interpret or compile a formal specification language, thus combining the advantages of a specification and a prototype. It seems clear enough that writing a specification/prototype is likely to be less work than writing a specification, writing a prototype, and verifying that the prototype matches the relevant part of the specification. REFINE [28], for example, provides much of the functionality of Z, but transforms its specifications into LISP. In this case, the cost of executability is a weakened set theory—all of REFINE's sets are finite, but Z supports

description of infinite sets—and some deviation from logical purity for the sake of efficiency. Also, use of an executable specification language tends to encourage over-specification, as a less abstract specification will execute more efficiently, providing a more useful prototype. Thus design decisions can creep into the specification, resulting in a commitment to design features that are appropriate for the prototype but inappropriate for the deliverable system. Again, the best approach is to explore the capabilities of the systems available at the time you read this, and decide for yourself whether any is well-suited to your needs.

Guideline 6.8 *Using an executable specification language entails the risk of over-specification, but offers substantial benefits. Whether the benefits balance the risks depends on the project and the choice of executable specification language.*

7. For Designers: Designing Your Software to Support V&V

7.1 Introduction

The principal advice we have for system designers is

Guideline 7.1 *To facilitate V&V, keep the design as simple as possible.*

In this section, the meaning of "simplicity" is spelled out in great detail, and the possibility of defining quantitative measures of design simplicity is explored. Also, the relative advantages of formal and informal design techniques are considered.

Note that, since design involves specification of parts, the material on specification in Section 6.2 is equally relevant to design.

7.2 What is Simplicity?

The notion of design simplicity seems, at first glance, to depend on the design method that is being employed. Consider, to begin with, classical design techniques based on functional decomposition. A good heuristic in using these function-oriented methods is: *Minimize the number of interconnections of subfunctions.* In other words, the design represented by the boxes and arrows—boxes are subfunctions and arrows represent data flow—in Fig. 7-1 should be preferred, all else being equal, to the design in

Fig. 7-2.

The criteria for judging the quality of an object-oriented software design seem to be quite different. The principal heuristic in object-oriented design is: *Let the messages to objects say "what" is to be done; associate "how" to do it with the most general classes possible.* Judged by the criterion for classical design, good object-oriented designs will fare rather poorly, as they tend to be highly interconnected.

Observe, however, that reducing the number of connections among subfunctions tends to minimize information flow among subfunctions. Indeed, reduction of information flow seems more fundamental than reducing the number of information channels, since reducing the number of channels simply by passing more complex data structures does not improve the design. And also note that the object-oriented design heuristic tends to reduce information flow across abstraction boundaries: many messages flow around the system, but each contains little information. Based on these considerations, the following seems to be a *generally* appropriate design heuristic.

Guideline 7.2 *No matter what design method you may be using, minimize information flow across module abstraction boundaries.*

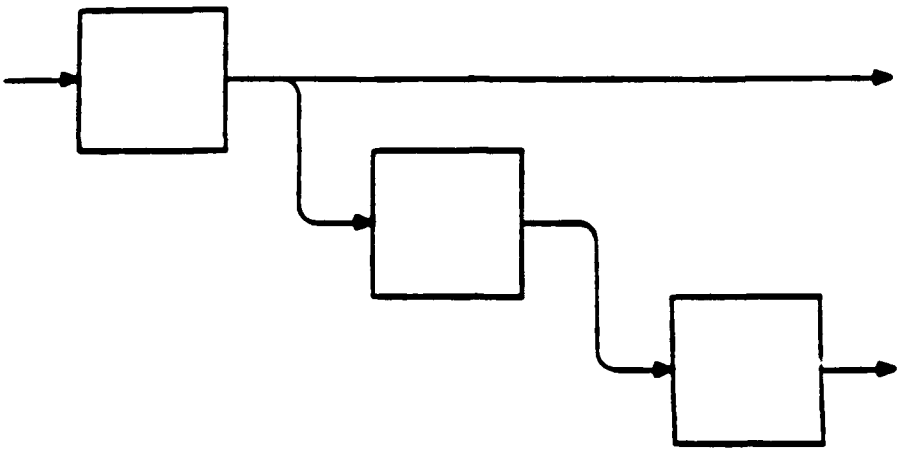

Figure 7-1: A Possible Functional Decomposition

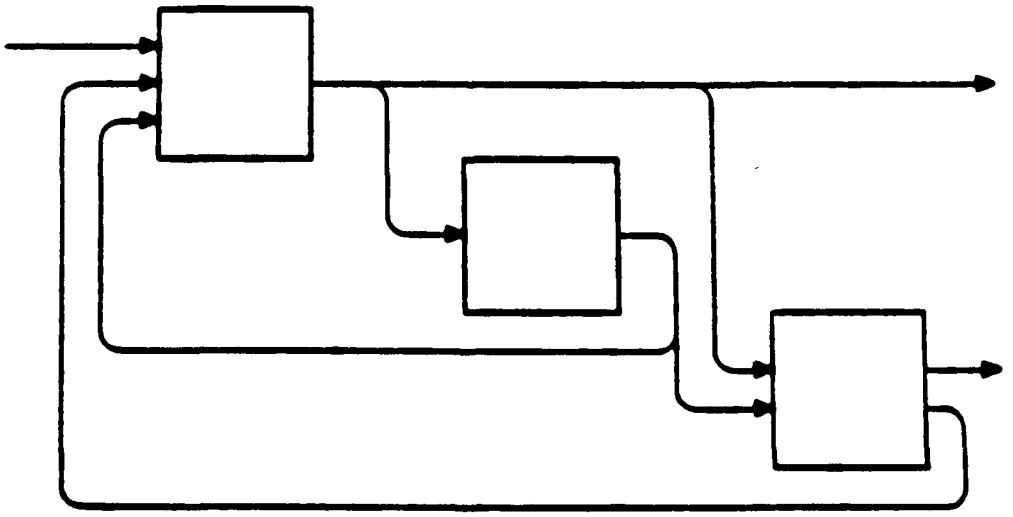

Figure 7-2: An Alternative Functional Decomposition

After all, one of the signs that you have identified a natural boundary is that the whole can be cleft relatively cleanly into parts at that boundary.

The consequences of using information flow as a measure of simplicity will now be considered.

7.3 Measuring Simplicity

Initially, consider the case of a knowledge-based system that calculates probabilities of hypotheses from evidence using a network of inference mechanisms that update probability distributions over their outputs based on changes to the probability distribution over their inputs. A typical system of this sort is shown in

Fig. 7-4.

Shortly after Shannon inaugurated the study of Information Theory, Carnap and Bar-Hillel [11] showed that a formal analogue of Shannon's information measure

$$\mathbf{I}(q) = -\sum_j q_j \log_2(q_j)$$

provides a natural explication of *amount of semantic information* when the probabilities q_j are interpreted as justified degrees of belief. Thus the amount of information associated with an inference mechanism that transforms prior probability distribution q on its output into posterior probability distribution q' is simply $\mathbf{I}(q') - \mathbf{I}(q)$. And so, given a design-level probabilistic model of the systems inferential behavior, the expected intermodule information flow can be computed.

The details of how to calculate this measure depend on the particular inference mechanism used. One might hope that in cases where the update algorithm is reasonably efficient (e.g., the recently developed network-based algorithms for Bayesian inference [32], or even more general algorithms for cross entropy minimization [38]) a reasonably efficient analytic determination of expected information flow would be possible. Note, however, that this calculation requires information not usually available prior to implementation: having all the knowledge required to actually perform inference when the design is completed is the exception rather than the rule in knowledge-based system development. Fortunately, expected information flow can always be estimated by simulation in a completely straightforward fashion at design-time using the high-level probabilistic model based on estimates of the average information added by each inference mechanism.

The next question that must be addressed is: *How can this be generalized to other sorts of systems?* If quasi-probabilistic mechanisms are used, the generalization is easy. But the generalization to systems that perform, say, deductive inference is not so easy. The problem is that, according to the standard definition of semantic information, deduction does not yield new information: if you assign A a probability of 1 and A implies B, then B must also be assigned a probability of 1 to maintain

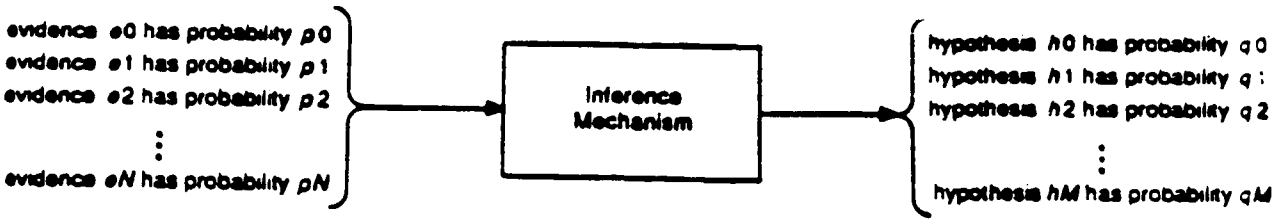

Figure 7-3: An Inference Mechanism

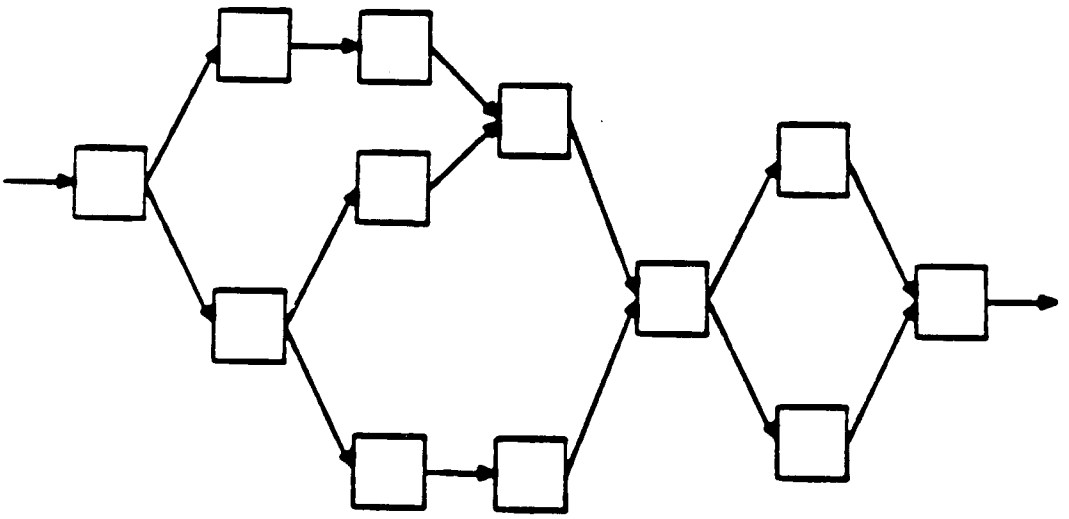

Figure 7-4: A Network of Inference Mechanisms

consistency. So what is needed is a sense of information such that if you initially know that A but not that B and you later succeed in deducing B from A, you have gained information. [1]

Thus, defining a quantitative measure of simplicity that can provide design-time guidance in the quest to support V&V is only a partially solved problem. But many AI systems are of the probabilistic sort where a useful measure has been already been defined.

Guideline 7.3 *If a simplicity measure is available, use it to evaluate the relative simplicity of radically different designs.*

7.4 Formal vs Informal Designs

If the sole purpose of documenting designs were to provide guidance to the implementors, then the defects of using an informal or semi-formal design method---natural language, Structured Analysis, Structured Design, and Commercial Products such as SADT, IDEF, etc.—may be balanced by the main advantage: any programmer can understand designs presented in these formalisms with little or no training. But when strong verification of the code against the design is required, the advantages of a more formal design dominate. In fact, since design is decomposing the system into parts, specifying those parts, decomposing those parts into subparts, and so on, the advantages of formal design are exactly the same as those of formal specification, already described in Section 6.2.

---

[1] We explored three different approaches to explicating this more generally applicable concept of information.

- Taking a *semantic* approach, we employed a broader class of models (including, say, urn models) that distinguish between non-trivially logically equivalent sentences, and left the definition of information remains the same.
- Taking a *syntactic* approach, we measured the complexity of the simplest derivation of B from A and used that as the measure of information gained in inferring B from A.
- Taking an *epistemic* approach, we used a "fine grained" epistemic logic and defined a new concept of information in terms of the old

$$\mathbf{I}'(A) = \mathbf{I}(\mathsf{Knows}(\mathsf{system}, A))$$

We then showed that the semantic and syntactic approaches are special cases of the epistemic approach, so our subsequent research focussed on the epistemic definition of information. (This one of a number of cases where mainstream AI techniques may have enormous impact on the V&V problems raised by AI.) At the time of the writing of this manual, the research is not sufficiently mature to provide any immediate guidance on how to measure information flow in non-probabilistic systems, but it does show considerable potential.

8. For Programmers: Choosing Programming Techniques

In this section, we discuss the impact of the choice of the AI programming techniques of Section 3 on the appropriateness and effectiveness of various V&V techniques. Guidelines for choosing programming techniques that simplify V&V are presented. Also, for each technique, we comment on the difficulty of providing a given level of verification, which verification techniques might usefully be employed, and how and what to test to provide convincing validation of the system.

Although this list of techniques is far from exhaustive, the main objective is to provide you with some examples of how the conventional wisdom of V&V can be applied to AI development, even though many AI programming techniques are never considered in V&V textbooks. If a programming technique of interest is not listed, you should be able to add it by using the following techniques as models.

8.1 Data-Driven Programming

Recall that the essence of data-driven programming is that data triggers the retrieval and running of associated programs. Typically, these techniques will be used in a situation where a specification specifically calls for a certain function to be performed for a certain independently-specified program to be executed when the data satisfies a certain condition. Thus, the "incremental" verification problem introduced by the use of data-driven programming is showing that the data-program association is correct according to the specification, and that the condition on the data is always tested when appropriate. (This observation explains why data-driven programming is such a useful technique. Saying something like "*Whenever the data satisfy condition C, then do A*" is a very natural way of specifying desired behavior. Verification of the code that tests the condition on the data is generally easy, as that code is simply an optimization of a naïve implementation of the condition, and the verification of the associated programs must be performed anyway. So use of data-driven programming—as opposed to more traditional techniques that do not treat the conditions and programs as "first-class objects"—is straightforward, because the structure of the code more closely matches the structure of the specification.)

So our first guideline on choice of programming techniques to support the V&V process is:

Guideline 8.1 *Use data-driven programming when, but only when, the form of the specification naturally calls for it.*

If this guideline is adhered to, the incremental validation problem can usually be made tractable, even if very strong formal validation is required. Validation of the data-program association should be trivial, whether the association is explicitly represented

in a data structure, implicitly represented by inclusion of a method definition within the lexical scope of a class declaration, or whatever. No matter what programming language mechanism is used to create the association, exactly what data is associated with what program should be clear enough. The potentially difficult part of the validation is assuring that the tests are always applied when appropriate. Probably the most straightforward solution is to use some sort of active data structures with a well-defined interface, so that the code for testing the condition on the data can be integrated with the code for changing the data. Most modern languages (LISP, C++, Ada, and so on) provide support for encapsulating data structures, so this straightforward solution can be easily implemented.

Guideline 8.2 *Encapsulate data structures that contain data with associated procedures, using the appropriate linguistic mechanism (if there is one).*

One prominent case when these techniques cannot be applied is when either (1) PROLOG is used and the data associated with programs is simply a subset of the PROLOG database, or (2) a shell for building rule-based systems is used, the data associated with programs is part of the database the rules operate on, and the database does not support attachment of daemons to the data. (For present purposes, a *daemon* is simply a procedure that can be associated with data that will execute when the data is changed. That is, it is a basic data management mechanism for supporting data-driven programming.) In this case, the best solution is to rewrite the program so that the data is not stored in the PROLOG database. If this is infeasible—because, for example, the data is being used in computational reflection—the next simplest solution is to distribute the data-program association among the individual PROLOG program clauses or production rules. For example, if the clause

$$P(x, y) \text{ :- } Q_1(x, z_1),\ Q_2(x, z_2), \ldots,\ Q_n(x, z_n).^1$$

might change the Prolog database so as to make condition C true, and if A is to be performed whenever C becomes true, the clause should be rewritten

$$P(x, y) \text{ :- } Q_1(x, z_1),\ Q_2(x, z_2), \ldots,\ Q_n(x, z_n),\ (\ C \text{ -> } A\ ).$$

The incremental verification problem then becomes assuring that the conditionals have been added everywhere they are required—which can require an arbitrarily complex analysis of which clauses might make conditions on the data true when they are called. This may not be difficult in a particular case, but the following guideline provides sound general advice.

[1] The use of :- for *if* and , for *and* is standard Edinburgh PROLOG notation. We will also use -> for the more-or-less standard conditional operator.

Guideline 8.3 *Avoid using data-driven programming when the data is stored in a globally accessible database that does not support attachment of daemons to the data.*

An alternative, in the case of PROLOG, is to build data-driven programming support into a meta-interpreter. This is an elegant solution, but typically far harder to verify than the rewritten rules.

Use of data-driven programming does not much affect system validation. When a data–program association is dictated by the requirements, the *correctness* of the data-program association will be adequately demonstrated in extended use. The principal added burden in validation is making sure that there are no cases when, given the data, some program should have been called, but was not. If the data can be monitored independently—by running under a debugger, for example—sampling values may raise the level of confidence in the *completeness* of the implementation of the data-program association.

8.2 Discrimination Nets

Discrimination nets are, in effect, an optimization of a specified more-or-less "flat" categorization. That is, the specification of the categories will probably look something like

> *If* $P_1(x)$, *then* x *is a* C_1. *If* $P_2(x)$, *then* x *is a* C_2. *If* $P_3(x)$, *then* x *is a* C_{31} *if* $Q_3(x)$, *else* x *is a* C_{32}. ...

A rather formal verification that the categorization implicitly defined by the tests in the net correctly correspond to the specified categories is usually feasible. The optimization used to build the tree amounts to observing that there are logical dependencies among the properties, so the results of previous tests can be used to simplify subsequent tests. What must be verified is that, if category C is associated with leaf node N, then the series $T_0(x) = r_0, T_1(x) = r_1, T_2(x) = r_2, \ldots$ of test results associated with any path leading to N implies that x is indeed C. Since the tests were chosen so as to have this property, it should be easy enough to show that they do.

Guideline 8.4 *When optimizing the tests in the net, consider not only run-time efficiency, but the difficulty of demonstrating correctness.*

One of the advantages of this programming technique is that a relatively limited set of test data can provide a great deal of confidence that the classification is correct. Generally, the level of confidence rises more quickly if the intermediate categories defined by partial test results are natural, rather than invented. More important,

revalidation is simplified after making the most common change to the net, adding new categories (including refining existing categories). This can be achieved by associating each category C with a collection $\{F_0, F_1, F_2, \ldots\}$ of independent primitive features that jointly guarantee membership in C, and then treating the optimization problem as being one of ordering the tests for the F_i. In the ideal case, each initial segment of the ordered list of features will determine a natural category, i.e., each test will determine which natural subcategory of a natural category the given information belongs to. (This approach will produce discrimination trees, rather than sometimes more efficient general nets.) In other words, the discrimination is based on a natural classification hierarchy.

Guideline 8.5 *When possible, base the discrimination on a natural classification hierarchy.*

8.3 Meta-Level Control Structures

The principal verification burden imposed by meta-level control mechanisms is that COMMON LISP programs containing undisciplined uses of `function` can be difficult to reason about using conventional specification formalisms. Typically, specifications will either include a very procedural representation of the algorithm to be used, or will be formalized more abstractly in a higher-order or set-theoretic specification language. Therefore, the problem can be minimized by using only closures that need not close over any variable bindings, in which case the closures can be treated as mathematical functions that contain no information about the environment. The same can be said, mutatis mutandis, about the use of SCHEME's procedures.

Guideline 8.6 *When implementing meta-level control structures, avoid using closing over functions that contain free variables, to simplify verification.*

A major advantage of using meta-level control structures, from the standpoint of V&V, is that some requirements on the functioning of the system might only be verifiable via proving—either formally or informally—that its control structure has certain desirable properties. For example, we might require that a system which contains several routines that might be applied in problem solving to select among them *fairly*. Proving such properties is often much easier when control structures are explicitly represented at the meta-level.

Guideline 8.7 *If verification requires proving that the program's control structure has certain properties, consider representing control explicitly using meta-level control structures to simplify verification.*

Validation of systems that make use of meta-level control can become more complex because there is no longer a meaningful distinction between data and control.

A formal requirement for, say, exhaustive branch testing cannot be satisfied, because new branches are created at run-time based on the data. From an unsympathetic viewpoint, standard implementations of meta-level control structures might be compared to self-modifying code—which is thought to be difficult-to-impossible to strongly validate—in this respect. Rather than relying on the usual criteria for determining the extent of validation, ad hoc arguments that the desired level of validation has been achieved will be necessary. More effort must be put into defining and defending validation procedures when meta-level control mechanisms are employed. (Those procedures may not be especially difficult or expensive to perform, however.)

Guideline 8.8 *Although standard criteria for comprehensive validation may be hard to satisfy, implementations with meta-level control structures are not necessarily difficult to validate (i.e. it frequently is not difficult to exercise such structures in ways that provide considerable confidence that they perform as they should).*

8.4 Deductive Information Retrieval

Deductive information retrieval, being based on formal logical deduction, can be another good candidate for formal verification. But, as was noted in the description of the technique, such systems typically deviate from "logical purity" in some way, and the extent and nature of the deviation can influence the efficacy of formal verification techniques. First, there may or may not be a mathematical semantics that determines whether derivations are correct or not. Second, if there is a formal semantics, the deductive technique may or may not be complete with respect to that semantics, i.e., the deductive technique might not be strong enough to derive all conclusions that follow from the facts stored in the knowledge base. Third, the deductive technique may or may not be sound with respect its formal semantics, i.e., the deductive technique might allow incorrect conclusions to be drawn from the knowledge base.

Guideline 8.9 *If the extent of the consequence relation is under your control, both verification and validation will be greatly simplified if you choose a relation based on a mathematical semantics, rather than one that can only be defined in procedural terms.*

Guideline 8.10 *Make the deduction technique as close to complete as performance constraints allow. And make sure that all obvious conclusions will be drawn from the knowledge base, because missing obvious conclusions will make the user doubt the utility of the system.*

Guideline 8.11 *Be extremely hesitant to give up on soundness. Even if you can argue that no incorrect conclusions will be drawn in practice, use of an unsound deduction technique will tend to undermine the user's confidence in the system, making validation and revalidation much more expensive and difficult.*

Whether a "nice" deductive technique exists is often determined by the specification. If the specification determines exactly what conclusions should follow from the facts in the knowledge base, the programmer must implement that particular notion of consequence, no matter how messy it may be. (Specifications sometimes underdetermine the consequence relation, because a good specifier knows that most standard notions of logical consequence are too expensive to implement efficiently and chooses to leave the details to the programmer's discretion.) Compromising logical purity for efficiency's sake is a practical necessity, and the desired effects of a compromise are usually impossible to express except in form of an algorithm. So most specifications will provide a procedural representation of the desired consequence relation, and the programmer's task is simply to implement that algorithm. In this, the most common, case, there is no special verification or validation problem associated with the deduction engine: it is simply a matter of showing that an algorithm has been implemented correctly. Moreover, some languages and shells—most notably PROLOG—provide an deduction engine which need not be verified and is already well validated.

Guideline 8.12 *Whether deductive information retrieval will be used should be determined by the form of the specification of the system's retrieval capabilities.*

There can be a second set of V&V problems associated with deductive information retrieval: the general knowledge in the knowledge base must be validated. (Verification is usually trivial. The general knowledge is given in the specification, in a form that can be directly coded in the knowledge representation formalism.) If, when the system is used, the knowledge base consists mainly of data, validation may be straightforward, with just a few carefully defined retrievals being sufficient to show that the general knowledge is correct. But typically, the general knowledge is extensive, consisting of hundreds or thousands of general facts. While there are classical techniques for verification and validation of such collections, such as showing that the knowledge base is consistent by showing that it has a model, and some progress has been made on defining analytical techniques to show that a knowledge base has other desirable properties [41], there is no substitute for extensive validation testing. In most cases, this is not a problem for the programmer, because the specification gives the (possibly buggy) general knowledge, but it is still worth noting at this point.

Guideline 8.13 *The power of deductive information retrieval is that the knowledge can be combined in unexpected ways to draw surprising conclusions. It is hard to guarantee all these conclusions will be correct, even when you can guarantee that they are all consequences of the general knowledge and the particular data. In other words, it is hard to validate knowledge bases.*

8.5 Production Systems

Production systems represent a more radical departure from conventional programming than the techniques discussed above, so much so that our notion of the "in-

cremental" verification and validation problems associated with a programming technique is not really meaningful in this case. There is an illuminating analogy that can be drawn with deductive information retrieval, however. The recognize-act loop can generally be verified using conventional techniques, but this code is usually quite simple compared to highly optimized constructive theorem prover. (This is not invariably true, however; some rule-based system shells provide very complicated rule selection mechanisms.) In addition, the recognize-act loop is usually "given" by the programming language or shell being used. At any rate, verification and validation of the recognize-act loop is not the difficult problem; verification and validation of the rules is.

The difficulty of verifying and validating a collection of assertions, discussed in the previous section, is even greater for rules. While a general assertion, *If it is true that A, then it is true that B*, may used as if it were a special case of a production rule, i.e., as *If it is true that A, add B to the knowledge base*, the verification techniques used for rules do not generalize in a straightforward fashion to arbitrary production rules. A rule's action-part can make an arbitrary change to some data structure. If that data structure in some sense represents external reality, then the change naturally corresponds to an assertion, and the verification techniques used for deductive information retrieval can be employed. But, as experience with PROLOG programming has demonstrated, it is not always possible to find a nice "declarative interpretation" of transformation rules.

Production rules can be used to perform steps in arbitrary algorithms, and it is generally the algorithm or the function computed by the algorithm that is found in the specification, not production rules. It may be easy to verify that a collection of rules *considered in isolation* is an implementation of some algorithm that can be verified against the specification, but the verification that, when combined with other rules, no problems will arise may be completely infeasible. After all, the point of the production system architecture is to encourage surprising interactions among the rules.

Guideline 8.14 *If you have a functional specification and strong V&V requirements, production systems are not a good candidate for implementation.*

Validation of knowledge bases is at least somewhat simplified in virtue of useful properties of deduction. For example, deductive inference preserves truth—if the premises are true, the conclusion must also be true—and hence is *monotonic*, i.e., new data added to the knowledge base cannot undermine previous conclusions. Without some such properties associated with the application of the production rules, verification and validation of a rule base are practically impossible.

Guideline 8.15 *Attempt to define invariants that capture all relevant requirements on the function implemented by the production system. If you can define such invariants, the same sort of techniques used in V&V of deductive information retrieval can be applied.*

8.6 Frame Databases

In contrast to the other programming techniques we have considered, the use of frame databases has no direct impact on the efficacy of V&V procedures. In fact, with the advent of CLOS—especially the CLOS Metaobject Protocol—COMMON LISP implementations of advanced features of frame databases tend to be simple and straightforward [44].

8.7 Backtracking

Whether backtracking presents any special verification difficulties depends on the type of backtracking and the specification. If the backtracking algorithm to be used is specified, there is no particular difficulty in verifying that it has been implemented correctly. But problems can arise when the algorithm is imprecisely specified – as we have already noted, many specification languages are not good at specifying complicated control regimens—or the algorithm used in the implementation is a "qualitative" optimization of the specified algorithm. An example of the latter is replacing naïve dependency-directed backtracking by more sophisticated reason maintenance. In this case, verification of the algorithm can involve very complicated reasoning about state. The difficulty of verifying the correctness of such an algorithm might be comparable to verifying the correctness of a clever garbage collection algorithm, that is, it stresses the present state-of-the-art.

Guideline 8.16 *More sophisticated backtracking techniques are much harder to verify than simpler techniques; use them only if efficiency demands it or comparatively weak informal verification is sufficient.*

Even complicated backtracking algorithms can be validated in a straightforward fashion, however. But, just as in the case of meta-level control structures, conventional validation requirements are somewhat inappropriate to cases where substantial meta-level reasoning about control is performed.

Guideline 8.17 *Although implementations of advanced backtracking techniques are not especially difficult to validate (i.e. it is not especially difficult to exercise them in ways that provide considerable confidence that they perform as they should), standard validation criteria may be hard to satisfy. If such criteria are externally imposed on the development effort, use the more primitive techniques, such as chronological backtracking.*

9. For Documenters: Documenting Your Efforts

It has often been asserted that DOD-STD-2167A is not really appropriate for AI development because, no matter what the standard may say, it is fundamentally based on a waterfall development model, while AI development is usually based on an iterative prototyping model. One of the principal theses of this manual is that 2167A is perfectly appropriate to AI development—in fact, that it provides excellent guidelines to help ensure that development will produce a well-engineered system if it is understood as requiring the development of certain lifecycle *products*, rather than dictating a particular lifecycle *process*. (See Section 5 for the arguments that requirements, specification, and design documents are as valuable when prototyping as when waterfall development models are employed.) The adaptation required to fit 2167A to AI development is essentially the same as that to fit it to Boehm's Spiral Model [6]—the requirements definition, system specification, system design, etc., must be treated as living documents that are revised and expanded throughout the lifecycle.

The reasons for documenting can be seen most easily by focusing on the maintenance phase of the lifecycle (although a change in personnel during development provides nearly as good an illustration). The software was written to solve some problem by performing its task. When the problem changes, as problems have a tendency to do, the software must be changed as well. If no formal record of requirements was generated, there is no documentary guidance available when attempting to decide whether making a change will interfere with the system's functioning. Will the change affect essential features of the implementation, or only accidents? If essential features are affected, are they affected in a significant way? The requirements statement should answer these questions by saying what the system must do in order to solve the problem. What the system actually does, how it does it, how we determine that it does what its supposed to do, and so on, are irrelevant to answering these questions; this document says what counts as a solution, not which solution was implemented. Therefore, the system specification, the system design documents, verification traces and validation suites, and so on, are irrelevant to the requirements statement.

Guideline 9.1 *The requirements statement can and should read as if it were written prior to the other documents, as it is in the waterfall model, even if it is the last document actually completed.*

Moreover, all other project activities—specifying the system, designing it, and so on—are influenced by the staff's current best understanding of the system requirements. For example, the system specifier must have some guiding idea of what the requirements are, for decisions on what the system *will* do are determined by the specifier's understanding of what it *must* do; the specifier's job is to choose and describe the specification that will be (or has been) implemented from among those that satisfy the requirements. Even if the requirements are not well understood at the time specification commences—the most common case in AI system development—it is well

worth writing the best current understanding of the requirements down, so that there will be a record of the basis of the specification. And, unless the development staff consists of a single person, efficiency demands that the specification be written down, to ensure a common understanding among the staff. And since the first project activity is an attempt to understand the problem and possible approaches to solving it, it makes sense that the first document produced should be a record the results of that activity.

Guideline 9.2 *The first draft of the requirements statement should be written at the beginning of the project, as a basis for all other lifecycle activities.*

But these same arguments can be applied, mutatis mutandis, to the system specification: the purpose of the specification is to describe the system as a "black box" —it describes the functionality of the system, its external interfaces, and so on—so none of the information in documents that are produced later in waterfall developments is relevant: all other activities—design, acceptance test definition, and so on, excepting only requirements definition—depend on the specification: it is well worth documenting the best current approximation to the full final specification, both as a record of presuppositions of other activities and to help ensure a common understanding among the project staff.

Guideline 9.3 *The system specification can and should read as if it were written after the requirements statement, and before most code development and the system specification should be updated throughout the development life-cycle. A first draft of the specification should be written at that time, as a basis for all subsequent lifecycle activities.*

The general conclusion should be clear at this point. Each document is intended to serve a certain purpose in subsequent development and maintenance. The ordering of these documents imposed by 2167A is really determined by the purpose of the documents, even though the ordering is that actually employed in waterfall developments. (This is no accident, of course. The waterfall model was suggested by an examination of the "natural" ordering of the processes—you can't specify a system unless you know the requirements, you can't design the system without knowing the specification, and so on—without regard to the fact that even the final stages of system validation can reveal additional requirements on the system which entail revision of every lifecycle product.) Guideline 9.3 can be generalized to:

Guideline 9.4 *Each document should be produced in initial draft form, in the order called for in* DOD-STD-2167A, *and should be updated throughout the development lifecycle. Every document should, in final form, read as if it had been the product of a waterfall development effort.*

Jointly, the documents serve as a sort of rational reconstruction of the development effort, a description of the order in which activities would have occurred if only we had known then what we know now.

10. Notes on Testing

Testing AI software is not radically different from testing any other type of software. The same general rules apply: *test on typical inputs, test around extremal values, test exhaustively (e.g., exercising all statements, exercising all branches) if feasible*, and so on. Section 8 contains comments about the feasibility of strong validation as a function of programming techniques. This section covers the few differences between testing AI software & testing other software.

First, it should be noted that extensive testing—from the subunit to the system level—usually plays a central role in AI development efforts. Debugging consists of testing in the context of an error, followed by changes to the code in preparation for retesting. AI programmers, especially LISP programmers, become very sophisticated at debugging during development, and good AI programming environments provide sophisticated debugging tools. The principal difference, from the viewpoint of V&V, between testing during debugging and validation testing is that the former is much less formal. Debugging typically uses ad hoc tests rather than a test suite derived from a specification, and eschews any recording of the results. (However, a number of studies have been made of heuristics for debugging AI programs.) Thus, one strategy for encouraging AI programmers to employ more comprehensive and systematic testing strategies is to create tools that make validation look more like debugging. The ASP tool described in Appendix A is one example of such a tool.

Second, there is a very important practical question that has not been addressed yet. While developing statements of requirements and specifications may be desirable from the standpoint of V&V, many AI development efforts proceed to develop code based on an informal understanding of the problem, together with feedback from users on a series of prototype systems. At the conclusion of the development effort, the only documentation of the system is the source code and some sort of user's manual. How can validation proceed under these conditions? The most promising techniques for improving testing in the absence of a good specification are automatic generation of test suites and so-called "program mutation" techniques.

Classically, automatic test suite generation has been based upon structural analysis of the program, since the program is the only sufficiently formalized representation of the intended behavior. If other lifecycle products are sufficiently formalized, tests can be generated from them as well. What the tests reveal about the software depends on the lifecycle product from which they were derived.

- Tests derived from requirements contribute directly to validation (provided there is good reason to believe the requirements have been stated accurately).
- Tests derived from the specification contribute directly to demonstrating correctness of the implementation, and hence indirectly to validation.

- Tests derived from the system design contribute directly to verification of the implementation.
- Tests derived from the code itself can demonstrate that the implementation is robust, that it is free of minor "typographical" errors, and so on.

All these sorts of testing are desirable, and are mutually complementary.

A good example of deriving tests from code is path analysis testing, a method whose scope and limits are well understood [23]. But only in special cases can an automatically generated test suite guarantee that errors are detectable: restrictions on the sort of error sought [8] or on the sort of program being tested [24] are required. The consensus of the community—as reflected in, e.g., the OSI Conformance Testing Methodology and Framework test case selection guidelines—is that expert judgment and experience are required to supplement informal heuristics.

In cases where an appropriate formal specification is available, fully automatic test suite generation may well be feasible. For example, communication protocols are often specified using finite state automata or Petri nets. For simple protocols, such as TCP's "three way handshake" for connection establishment, path testing techniques supplemented by a formalization of the OSI heuristics can be used to mechanically generate a reasonable conformance test suite.

Program mutation techniques [10], on the other hand, can be usefully applied in most projects. Given only relatively weak restrictions on the program [20], mutation testing can reveal all "typographical" errors in the program. Such errors are usually among the most frequently made and difficult to detect: good programmers generally understand the algorithms well and design correct implementations, but are not immune to simple coding errors, such as being off-by-one on a loop exit condition. Developing automated support for mutation testing is clearly feasible, based on the analogy to genetic algorithms, although no attempts to do so have been documented in the literature.

Bibliography

[1] L. Aiello and G. Levi, "The uses of metaknowledge in AI systems," *Proceedings ECAI-84*, 1984, pp. 707–717.

[2] J. Allen, *Anatomy of LISP*, McGraw-Hill, 1978.

[3] Y. Bar-Hillel, *Language and Information*, Addison-Wesley, 1964.

[4] M. Bidoit, M. Gaudel, and A. Mauboussin, *How to make specifications more understandable? An experiment with the PLUSS specification language*, Rapport de Recherche 343, Centre d'Orsay, Université de Paris-Sud, 1987.

[5] D. Bjorner and C. Jones, *Formal Specification & Software Development*, Prentice-Hall, 1982.

[6] B. Boehm, "A Spiral Model of Software Development and Enhancement," *IEEE Computer* **21**(5), May 1988, pp. 61–72.

[7] I. Bratko, *Prolog programming for Artificial Intelligence*, Addison-Wesley, 1986.

[8] M. Brooks, *Determining correctness by testing*, Report STAN-CS-80-804, Department of Computer Science, Stanford University, May 1980.

[9] W. L. Bryan and S. G. Siegel, *Software Product Assurance: Techniques for Reducing Software Risk*, Elsevier, 1988.

[10] T. Budd, R. DeMillo, R. Lipton, and F. Sayward, "Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs," *Proceedings of the Seventh ACM Symposium on Principles of Programming Languages*, 1980)

[11] R. Carnap and Y. Bar-Hillel, *An outline of a theory of semantic information*, Technical Report 247, Research Laboratory in Electronics, M.I.T., 1952. (Reprinted in [3].)

[12] K. M. Chandy and J. Misra, *Parallel Program Design, A Foundation*, Addison-Wesley, 1988.

[13] E. Charniak, C. Riesbeck, D. McDermott, and J. Meehan, *Artificial Intelligence Programming*, 2nd edn, Lawrence Erlbaum, 1987.

[14] W. Clancey, "The advantages of abstract control knowledge in expert system design," *Proceedings of AAAI-83*, 1983, pp. 74–78.

[15] B. Cohen, W. T. Harwood, and M. I. Jackson, *The Specification of Complex Systems*, Addison-Wesley, 1986.

[16] R. Davis, "Meta-rules: reasoning about control." *Artificial Intelligence*, **15** (1980), pp. 179–222.

[17] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification*, Springer-Verlag, I:1985 II:1989.

[18] J. Goguen, "Parameterized programming," *IEEE Transactions on Software Engineering* **SE-10**, 1984, pp. 528–543.

[19] F. van Harmelen, "A classification of meta-level architectures," in [25].

[20] J. Gourlay, "A mathematical framework for the investigation of testing," *IEEE Transaction on Software Engineering* **SE9**(6), November 1983.

[21] C. C. Green, *et al.*, *Report on a knowledge-based software assistant*, RADC TR 83-195, Rome Laboratories, 1983.

[22] I. Hayes (ed.), *Specification Case Studies*, Prentice-Hall, 1987.

[23] W. E. Howden, "Reliability of the path analysis testing strategy," *IEEE Transactions on Software Engineering* **SE2** (3), September 1976.

[24] W. E. Howden, "Algebraic program testing," *Acta Informatica* **10**(1), 1978.

[25] P. Jackson, H. Reichgelt, and F. van Harmelen (eds.), *Logic-Based Knowledge Representation*, MIT Press, 1989.

[26] L. N. Kanal and J. F. Lemmer (eds.), *Uncertainty in Artificial Intelligence*, North-Holland, 1986.

[27] T. A. Linden, "Alternative Approaches to V&V for AI Systems," *AAAI Workshop on Validation and Testing of Knowledge-Based Systems*, August 1988.

[28] T. A. Linden and L. Z. Markosian, "Transformational Synthesis Using REFINE," in [35].

[29] T. A. Linden and S. Owre, *Verification and Validation of AI Software*, Technical Report TR-3209-02, Advanced Decision Systems, 1988.

[30] Theodore A. Linden, "A Meta-level Software Development Model that Supports V&V for AI Software," *Expert Systems with Applications: An International Journal*, Pergamon Press, 1.4, Nov. 1990

[31] M. Minsky, "A framework for representing knowledge," in [47].

[32] R. E. Neapolitan, *Probabilistic Resaoning in Expert Systems: Theory and Algorithms*, Wiley, 1990.

[33] P. Norvig, *Paradigms of AI Programming: A Common Lisp Approach*, Morgan Kaufmann, forthcoming (1991).

[34] M. Page-Jones, *The Practical Guide to Structured Systems Design, Second Edition*, Yourdon Press, 1988.

[35] M. Richer (ed.), *AI Tools and Techniques*, Ablex Press, 1989.

[36] W. W. Royce, "Managing the development of large software systems," *Proceedings of WESCON-70*, August 1970.

[37] R. Shank and R. Abelson, *Scripts, Goals, Plans, and Understanding*, Lawrence Erlbaum, 1977.

[38] J. E. Shore, "Relative entropy, probabilistic inference, and AI," in [26].

[39] B. Smith, *Reflection and Semantics in a Procedural Language*, Laboratory for Computer Science Technical Report 727, MIT, Cambridge, MA, 1982.

[40] J. M. Spivey, *The Z Notation: A Reference Manual*, Prentice-Hall, 1989.

[41] R. A. Stachowitz and J. B. Combs, "Validation of Expert Systems," *Proc. Twentieth Hawaii International Conference on System Sciences*, 1987.

[42] L. Sterling and E. Shapiro, *The Art of Prolog*, MIT Press, 1986.

[43] I. Van Horebeek and J. Lewi, *Algebraic Specifications in Software Engineering: An Introduction*, Springer-Verlag, 1989.

[44] J. Veitch, "Frames in CLOS," *AI Expert*, June 1991, pp. 41–47.

[45] J. Vincent, A. Waters, and J. Sinclair, *Software Quality Assurance, Volume I: Practice and Implementation*, Prentice-Hall, 1988.

[46] D. Warner Hasling, "Abstract explanations of strategy in a diagnostic consultation system," *Proceedings of AAAI-83*, 1983, pp. 157–161.

[47] P. Winston (ed.), *The Psychology of Computer Vision*, McGraw-Hill, 1975.

Part III

APPENDICES

A. ASP Manual

A.1 Introduction

ASP is an acronmy for "A Software Planner." It is a software tool to be used by programmers and software test personnel for planning, organizing, and executing the debugging, testing, verification, and validation of software systems.

A.1.1 Motivation

ASP is designed to reduce the problems involved in validating software that changes. Many modern software systems have very complex requirements that cannot be fully defined in advance. These systems are frequently developed with a rapid prototyping methodology, and they require extensive software maintenance and enhancements even after they become operational. Traditional testing, verification, and validation methods assume that changes to the software are infrequent and carefully controlled because changes require costly retesting, reverification, and revalidation each time the software does change.

Much AI software must adapt to evolving requirements, and this adaptability makes AI software difficult to validate. ASP is a software tool that reduces the cost and improves the effectiveness of retesting, reverification, and revalidation for AI software—or for any other frequently changing software.

A.1.2 ASP as a Software Tool

ASP allows programmers and software testers to specify and selectively control the execution of software test routines. The test routines are written as executable specifications-as predicates expressed either in the programming language in which the software is written or in a compatible executable specification language. These executable specifications are stored separately from the code, and ASP associates them with appropriate test points in the software implementation.

These tests or executable specifications are called verifications, and ASP makes it easy for a programmer to define these verifications early in the software development life cycle. These verifications are often more stable than the object software. By defining the verification early, the programmer can use them for debugging the software and also make them available to the personnel involved in the testing, documentation, verification, and validation phases of software development.

ASP gives the programmer full control over when the verifications are executed. Since execution of the verifications is frequently time consuming, ASP includes a declarative language for specifying which verifications are to be executed under what circum-

stances. These declarative specifications of what verifications are to be executed are called validations. A validation is a plan for controlling the execution of the object software while conditionally executing verifications that monitor, test, and verify the software as it is executing.

Executions of these validations or software plans are useful in all of the following roles:

External Debugger - ASP can help find bugs without perturbing the code being tested.

Procedural Debugger - Complex debugging scenarios can be easily devised in the Software Plan by specifying the conditions under which additional verifications are to be executed.

Validation - Validation suites are organized in the Software Plan.

Verification - Traditional correctness assertions in the code are replaced by tests in the Software Plan outside of the code.

Code Instrumentation - The programmer can specify in the Software Plan ways of visualizing how the code is performing and what it is doing in various circumstances.

ASP is an integrated tool that supports debugging, testing, verification, and validation. Part of the idea being supported by ASP is that it is useful for programmers to write verifications during the early stages of software development because these verifications can then be reused throughout the software life cycle. These verifications should not involve modifying the code that directly implements required software functionality; and, for performance reasons, these verifications should be executed selectively to meet the different goals of debugging, testing, validation, and operational execution.

The ASP software plans give the programmer control over the execution of software verifications without changing the software being tested or instrumented. ASP does not require recompilation before executing a different validation or software plan, and ASP tries to shorten rather than lengthen the programmer's cycle of executing, debugging, modifying, and re-executing the software.

ASP is designed so dependencies on the programming lanaguage in which the object software is written are isolated. The current ASP tool works for programs written in Common Lisp and handles executable specifications written in either Common Lisp or REFINE. It will be relatively easy to extend ASP to work with software written in other languages that allow function calls to be intercepted and redirected without recompilation. More effort will be required to make ASP work efficiently for languages like C and Ada where function calls do not involve a level of indirection.

A.1.3 A Software Planning Methodology

ASP can be used with any software development methodology. It is intended to reduce the validation problems associated with evolving software, and this subsection describes a general software methodology that is appropriate for the kinds of evolving software systems which ASP was designed to support. This software planning methodology is depicted in Figure A-1.

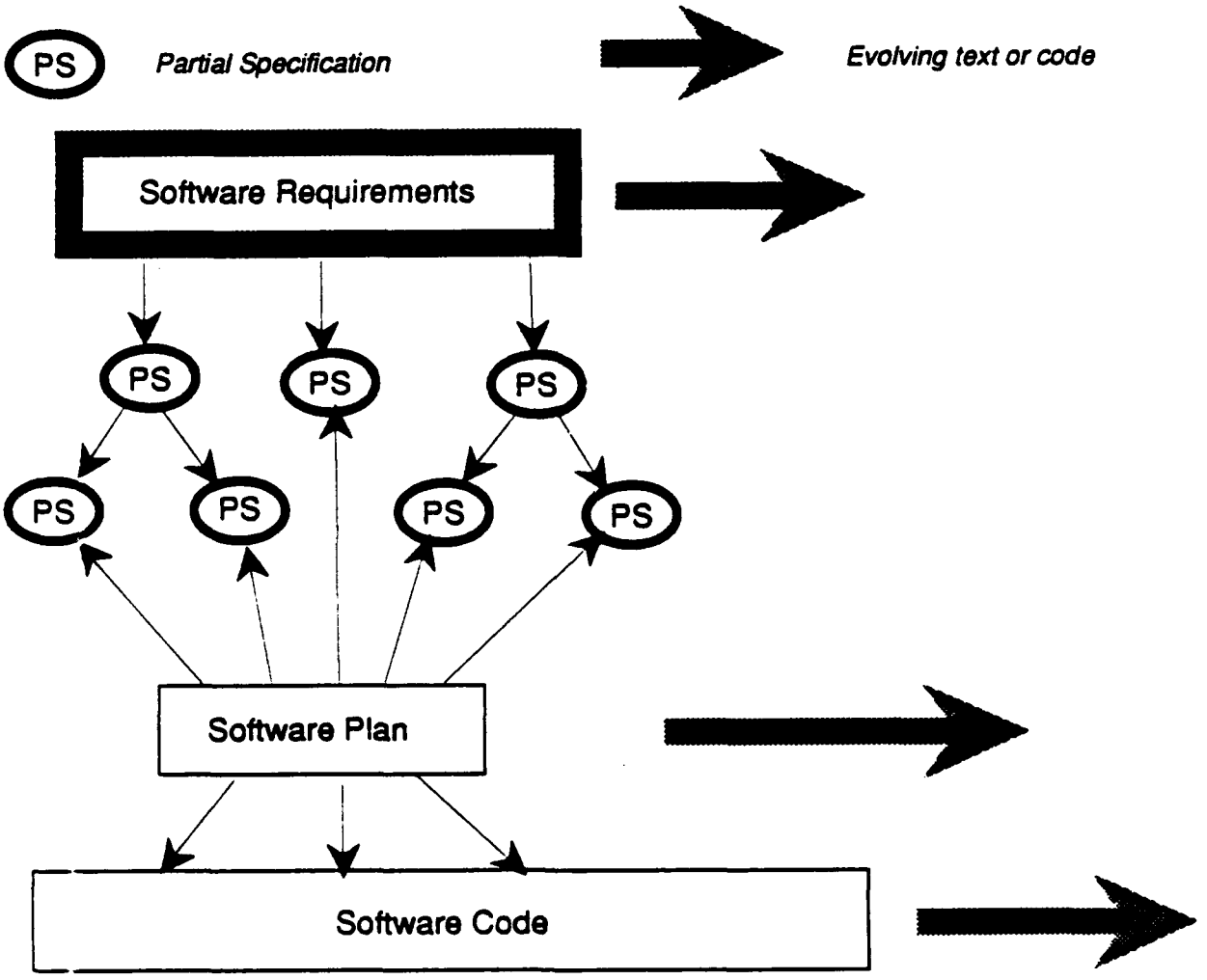

Figure A-1. Developing Evolving Software

In the early stages of software development, an initial version of the software requirements should be defined. Then these requirements, the software specification, and the software code are evolved concurrently—with the software plan used to interrelated the evolving specification and the evolving code.

Rarely is a software implemented in one phase from the software requirements. This is especially true of large or AI software systems. It is natural to develop it in bits and pieces over time combining smaller pieces into larger ones or stubbing out smaller pieces and adding them in progressive steps. With the software planning methodology supported by ASP, one tries to tighten the focus on this process by having the development of these bits and pieces driven by *partial specifications*. In *Figure A-1* a partial specification is depicted by a **PS** in an oval.

A partial specification is a statement of truth about the results of some partial computation of a software implementation. The collection of all partial specifications in this methodology represents the current software requirement. When all of the partial specifications in this collection are true we say that the current software requirement has been satisfied and we are ready to go on to the next phase of development. For example, suppose the current software requirement is that we construct a program that builds a table of the first n prime numbers. We could create three partial specifications:

1. A table exists with n entrys.
2. Every entry in the table is a prime number.
3. The n entries are the first n prime numbers.

When all three of these partial specifications are true then we have achieved the current software requirement. Partial specifications 2 and 3 could have been stated as one partial specification but it may have been easier to test for them as two weaker statements. In Figure A-1 this is depicted as a **PS** splitting off into two **PS**s. For the same reasons that it is easier to break programs into smaller pieces to debug them. it is easier to break program specifications into smaller pieces to test them.

The **Software Plan** in Figure A-1 ties these partial specifications to the **Software Code** and a decision can be made by the Software Plan that the current software requirement has been met. With ASP, using the prime number example, you have semi-automated this decision by displaying the table and visually inspecting the table to make sure that every number is a prime number. But you could also have written a predicate function that computes that every number in the table is in fact a prime number. We call such a predicate function an *executable specification*. In many cases it is actually easier to write an executable specification than it is to visually verify. In very complex software it is perhaps the only way to verify. The art of using this software planning methodology rests in being able to write a minimum number of executable specifications such that to a large extent the software verifies itself.

A.1.4 Using ASP

To use ASP one must supply the shaded boxes in *Figure A-2*, that is, the boxes labeled **ASP Software Plan** and **Verifications.**

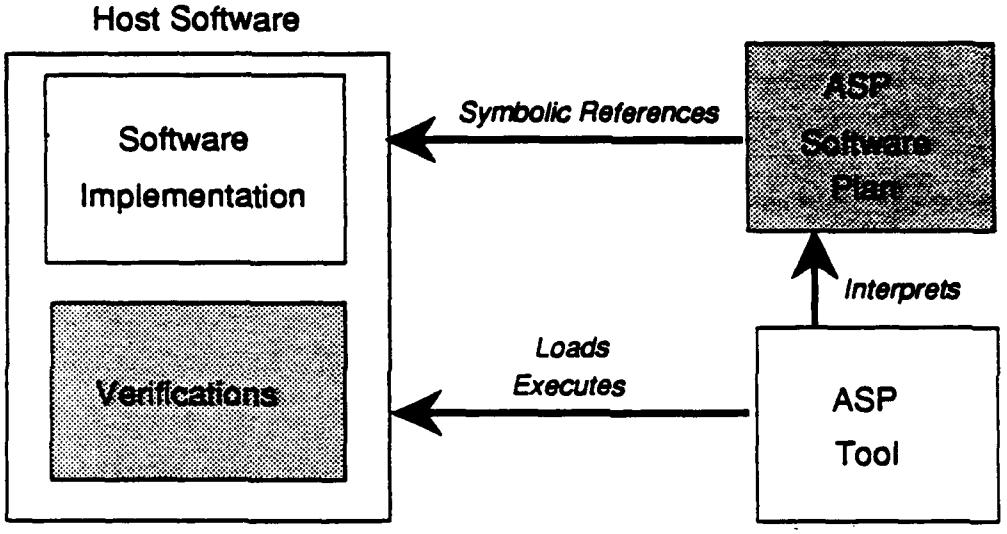

Figure A-2: ASP Components

The box labeled **Software Implementation** represents the software implementation before ASP is introduced. To use ASP nothing in the software implementation needs to change. All code in the box labeled **Host Software** is written in the *host language*. For example, if your software implementation is coded in Common Lisp then the verifications would be coded in Common Lisp. Verifications are usually just simple predicate functions. In the last section we described an executable specification which could be represented as such a predicate. In general an ASP *verification* is any function that supports verifying the results of computations in the software implementation.

The box labeled **ASP Software Plan** represents a text file that you configure. How to write a Software Plan will become clear in the following Find Word Example. Within the software plan you specify how you expect the software to perform. You do this by building *condition-action* trees in the Software Plan. At the top level these condition-action trees are called *validations*. Within the Software Plan are symbolic references to program objects in the software implementation and verifications. Since all of the references *are* symbolic, new implementations and verifications can be loaded dynamically, and ASP can do the right thing without recompiling the Software Plan.

We are now ready to talk about the box labeled **ASP tool.** To use ASP the first thing that you do is write a Software Plan which attempts to demonstrate the results of the current phase of software development. You then load the Software Plan and run the ASP Tool. The ASP Tool recognizes the loaded plan as the *current* Software Plan. Most of what the Tool does is based on interpreting the Software Plan. You

then select various validation scenarios depending on the desired effect. During the interpretation of any validation, ASP may load implementations and verifications and control execution of the host software depending on conditions in the validation.

A.1.5 Learning by Example

We use the method of teaching by example. We will first give a simple succinct example of using ASP, then we will explain how to use ASP in general. Since this example is delivered with ASP it might be helpful to actually load its Software Plan and interactively follow the example. A following chapter will describe the complete semantics of the Software Plan. Section A.6 has the complete syntax of the Software Plan. Examples of using some of ASP's more esoteric features are given later.

A.2 Find Word Example

The example Host Software we use is called Find Word. It implements a search algorithm in Common Lisp. Using this algorithm Find Word will find the number of occurrences of a given word in some given text. For example the word *sentence* occurs 3 time in the text "*This sentence is an example sentence for finding the word sentence.*"

We will first describe the algorithm and write the implementation. Then we will write a Software Plan to show that the implementation performs the way that we expect. Although this is not a very complex example, it will illustrate the idea of writing an ASP Verification, testing the software with multiple implementations, then debugging the software given that it fails the test. This example also illustrates the importance of executable specifications because in this case one ASP Verification that we write will represent an executable specification.

A.2.1 Writing the Find Word program

The intuitive approach to find a word in some text is to search for the first letter of the word and if found then compare the second letter in the word with the next letter in the text and so on. This is called a linear search. It turns out that we can search faster than that by using a sublinear search algorithm.

What we do in a sublinear search is

1. First look at the character at the words length into the text.
2. If this character is the last character in the word search backwards in the text to see if we are on the word.
3. If this character is another character in the word we skip ahead in the text by the same number of characters from where the character is in the word (closest to the end of the word) to the end of the word. And continue the search.
4. Otherwise we skip ahead the length of the word. And continue the search.

For example if we are looking for the word "sentence" in the text above

```
        "This sentence is an example sentence for finding the word sentence"
                 - - -
skip points:    1 2  3
```

First we go into the text to skip point 1, the length of the word "sentence". Since there is an "n" there we skip ahead to point 2 where the end of the word "sentence" would be if its character "n" was in that position. Since there is an "e" we search backwards and discover that we are not sitting on the last character of "sentence," so we skip to point 3 for the same reason that we skipped to point 2. Now we search backwards and find a match.

This skipping effect produces a faster search. We use sublinear search here not for its efficiency per se but because sublinear search needs a skip table set up for a given word prior to doing the search. We can specify what the skip table is to look like given any word. And in fact we write a predicate function that tests for what we specify. And we will refer to it in the ASP software plan. In ASP terminology we call this test predicate a Verification.

A.2.2 Find Word Code

First, we write the code called **WORD-SEARCH** to implement Find Word using sublinear search. This code was adapted from the book *How to Solve it by Computer* by R. G. Dromey. We also write a top level function called **COUNT-WORD** that calls **WORD-SEARCH** just so that we can print out the results. For this example it is not important to understand the code; just notice that we call **SETSKIPS** to set up the skip table.

```
(defun COUNT-WORD (word text)
  (let ((n (word-search word text)))
    (format t "~%The word ~s occurs ~a times in the text ~%\"~a~a" word n
            (if (< (length text) 101) text (subseq text 0 99))
            (if (< (length text) 101) "\"" " ..."))
    (values)))

; Given a word and text find the occurrences of word in text using
; a sublinear search algorithm.
(defun WORD-SEARCH (word text)
  (let ((wlength (length word)) (tlength (length text))
        ;;; Set up skip table.
        (skip (SETSKIPS word (make-array 128)))
        (nmatches 0))
    ;;; The i index skips through the text to the character nxt.
    (do ((i (1- wlength))) ((> i tlength) nmatches)
        (let ((nxt (char-code (aref text i))))
          ;;; Use skip table to drive search for pattern.
          (if (> (aref skip nxt) 0) (setq i (+ i (aref skip nxt)))
            ;;; Skip table indicates maybe at end of word so match backwards.
            (let ((j (1- i))
                  (k (- wlength 2))
                  (match t))
              (do () ((or (< k 0) (not match)))
                  (cond ((eql (aref text j) (aref word k))
                         (decf j) (decf k))
                        (t (setq match nil))))
              (when match (incf nmatches))
              ;;; Skip in the text to where the end of the word would be
              ;;; based on the character nxt.
              (setq i (- i (aref skip nxt))))))))))
```

```
; Set up the skip table of all ascii characters for the given word.
(defun SETSKIPS (word skip)
  (let ((wlength (length word)))
    ;;; The default skip for all characters not in the word is the word length
    (dotimes (i (length skip)) (setf (aref skip i) wlength))
    ;;; Otherwise the skip is determined by the character's
    ;;; position in the word.
    (do ((j 1 (1+ j))) ((> j (- wlength 2)))
        (setf (aref skip (char-code (aref word j))) (- wlength j 1)))
    ;;; assign negative skip to last character to differentiate from others
    (let ((p (char-code (aref word (1- wlength)))))
      (setf (aref skip p) (- (aref skip p))))
    skip))
```

A.2.3 Find Word Trials

To check our code we create two trials called TRIAL1 and TRIAL2. Actually these trials are well engineered for this example so that the first one will produce the correct answer and the second one will produce an incorrect answer (for our first implementation of WORD-SEARCH) because of two extra spaces between "This" and "sentence".

```
(defun TRIAL1 ()
  (count-word
   "sentence"
   "This sentence is an example sentence for finding the word sentence"))

(defun TRIAL2 ()
  (count-word
   "sentence"
   "This   sentence is an example sentence for finding the word sentence"))
```

A.2.4 Find Word Verification

To verify that the skip table is set up as we specified we create the test predicate, or verification, that we mentioned earlier. You might notice that SKIPTABLE-CORRECT is about the same size as the function SETSKIPS. Don't let this discourage you against using ASP to do program verification. This is apparent because the example is so simple. As the complexity of an implementation goes up, the size of the verifications goes down in proportion to the length of the implementations code; consequently the return in investment becomes greater.

```
(defun SKIPTABLE-CORRECT (word table)
  "For every character in Word there is a corresponding entry in the
skiptable that skips relative to the word's end character position. The
entry that corresponds to the last character position is a minus skip. All
other entries contain a skip equal to the length of the word."
  (let ((predicate t)
        (word-max (1- (length word))))
    (dotimes (i (length table))
             (let ((word-char-position (position (code-char i)
                                                 word :from-end t)))
               (unless
                (if word-char-position
                    (if (= word-char-position word-max)
                        (minusp (aref table i))
                      (= (aref table i) (- word-max word-char-position)))
                  (= (aref table i) (length word)))
                (setq predicate nil))))
    predicate))
```

A.2.5 Find Word Software Plan

Finally we write a Software Plan for Find Word.

```
(asp:specify-plan
 :name 'find-word
 :specifications
 '((skip-table-specification
    "/systems/asp/examples/find-word/verify"))
 :implementations
 '((find-word-sublinear-implementation
    "/systems/asp/examples/find-word/sublinear"
    "/systems/asp/examples/find-word/trials")
   (new-setskips-implementation
    "/systems/asp/examples/find-word/new-setskips"))
 :executables '((trial1) (trial2))
 :verification-points '((setskips word :output skiptable))
 :verifications '((skiptable-correct) (list))
 :validations
 '((LOAD-FIND-WORD
    ((:load find-word-sublinear-implementation skip-table-specification)
     (:on-entry :report) (:on-exit :report)))
   (TRIAL1
    ((:execute trial1)))
   (TRIAL1-VERIFY
    ((:execute trial1) (:on-entry :report) (:on-exit :report)
     ((:after setskips)
      ((:test (skiptable-correct word skiptable))
       (:on-pass :report) (:on-fail :report)))))
   (TRIAL1-DEBUG
    ((:execute trial1) (:on-entry :report) (:on-exit :report)
     ((:after setskips)
      ((:test (skiptable-correct word skiptable))
       (:on-fail :report
                 ((:record (list word skiptable) skiptable-snapshot)))))))
   (TRIAL2
    ((:execute trial2)))
   (LOAD-NEW-SETSKIPS
    ((:load new-setskips-implementation) (:on-exit :report)))))
```

Reading the plan from top to bottom: Its name is **find-word**. It has one specification called **skip-table-specification** which refers to the file verify.cl which contains the SKIPTABLE-CORRECT function. It has two implementations. One called **find-word-sublinear-implementation** which contains the code we defined earlier for the implementation of sublinear search and its trials. Another called **new-setskips-implementation** we will use later to fix a problem in the original code. It specifies that there are two possible execution point functions called **trial1** and **trial2**.

There will be one verification point called **setskips** whose input arguments will get bound to the Plan scoped variable **word** and whose output argument will get bound to the Plan scoped variable skiptable. There is one verification specified called **skiptable-correct** that we mentioned earlier and another verification **list** which is just the common lisp function LIST. A test (or executable specification) must be a verification, but a verification does not need to be a test, for example in this case the verification **list** is just used to record a list of outputs as one object. Finally for this example we choose to have six validations called: **LOAD-FIND-WORD**, **TRAIL1**, **TRIAL1-VERIFY**, **TRIAL1-DEBUG**, **TRIAL2**, **LOAD-NEW-SETSKIPS**.

A.2.6 Using the ASP tool

Now that we have defined a Software Plan for the Find Word software we are ready to run the ASP tool and run our software through the trails. First you would load the ASP system (see site specific README file for how to do this), and then load the Find Word Software Plan.

In the lisp listener we evaluate the following

```
(asp:tool)
```

After doing this we will see

```
Select ASP activity for FIND-WORD plan:
 0 = Exit ASP tool.
 1 = Select a plan validation for execution.
 2 = View PO attributes.
 3 = Change plan defaults.
Enter a number from 0 to 3 -> 1
```

At this point we enter a 1 to **Select a plan validation for execution.** We would then see

```
Select and execute one of the plan validations:
 0 = Execute no validation
 1 = LOAD-FIND-WORD
 2 = TRIAL1
 3 = TRIAL1-VERIFY
 4 = TRIAL1-DEBUG
 5 = TRIAL2
 6 = LOAD-NEW-SETSKIPS
Enter a number from 0 to 6 -> 1
```

Notice that the names of the 6 validations that we specified in the Software Plan now appear as selectable validations. We will first choose 1 to load the Find Word software. It is not necessary to have such a **Load Software** validation in your software plan, you may load your software independently of ASP. But it is included here to illustrate two things:

1. Loading software in itself can be considered to be a validation
2. When you get involved with more complex Software Plans that load many implementations over the course of different tests, you frequently want to have a Validation that simply reloads the base implementation.

After entering 1 we would see the following appear:

```
-------------------------------------------------------------------------
| ASP controlling FIND-WORD plan.                                       |
| Loading: FIND-WORD-SUB-LINEAR-IMPLEMENTATION                          |
-------------------------------------------------------------------------


-------------------------------------------------------------------------
| ASP controlling FIND-WORD plan.                                       |
| Loaded: FIND-WORD-SUB-LINEAR-IMPLEMENTATION                           |
-------------------------------------------------------------------------


-------------------------------------------------------------------------
| ASP controlling FIND-WORD plan.                                       |
| Loading: SKIP-TABLE-SPECIFICATION                                     |
-------------------------------------------------------------------------


-------------------------------------------------------------------------
| ASP controlling FIND-WORD plan.                                       |
| Loaded: SKIP-TABLE-SPECIFICATION                                      |
-------------------------------------------------------------------------
```

Looking back at the Software Plan notice that in the **LOAD-FIND-WORD** Validation we told it to report the results on entering and exiting the load process for `find-word-sublinear-implementation` and `skip-table-specification` and the above output shows that it did just that.

```
(LOAD-FIND-WORD
  ((:load find-word-sublinear-implementation skip-table-specification)
   (:on-entry :report) (:on-exit :report)))
```

At this point we will see the top level menu again. For the rest of what follows we will not repeat this, but assume that you will know that you start from the top level menu.

```
Select ASP activity for FIND-WORD plan:
 0 = Exit ASP tool.
 1 = Select a plan validation for execution.
 2 = View PO attributes.
 3 = Change plan defaults.
Enter a number from 0 to 3 -> 1

Select and execute one of the plan validations:
 0 = Execute no validation
 1 = LOAD-FIND-WORD
 2 = TRIAL1
 3 = TRIAL1-VERIFY
 4 = TRIAL1-DEBUG
 5 = TRIAL2
 6 = LOAD-NEW-SETSKIPS
Enter a number from 0 to 6 -> 2
```

This time we select 2 for the **TRIAL1** validation. Looking at the Software Plan we see that this Validation does nothing more than just execute the function called `trial1` with no other conditions. It is as if we ran the program without ASP being around and it produces the output of the Find Word program that it would if `trial1` were run by itself:

```
The word "sentence" occurs 3 times in the text
"This sentence is an example sentence for finding the word sentence"
```

This output looks correct and in fact if the programmer saw this he might assume that his program is working fine. But now we will try **TRIAL1-VERIFY** Validation:

```
Select and execute one of the plan validations:
 0 = Execute no validation
 1 = LOAD-FIND-WORD
 2 = TRIAL1
 3 = TRIAL1-VERIFY
 4 = TRIAL1-DEBUG
 5 = TRIAL2
 6 = LOAD-NEW-SETSKIPS
Enter a number from 0 to 6 -> 3

-------------------------------------------------------------------------------
| ASP controlling FIND-WORD plan.                                             |
| Entering: TRIAL1                                                            |
-------------------------------------------------------------------------------



-------------------------------------------------------------------------------
| ASP controlling FIND-WORD plan.                                             |
| Verification SKIPTABLE-CORRECT : AFTER point SETSKIPS.                      |
| Result = ***FAILED***                                                       |
-------------------------------------------------------------------------------

The word "sentence" occurs 3 times in the text
"This sentence is an example sentence for finding the word sentence"

-------------------------------------------------------------------------------
| ASP controlling FIND-WORD plan.                                             |
| Exiting: TRIAL1                                                             |
-------------------------------------------------------------------------------
```

This time instead of just the program output we see that ASP is now engaged.

Looking at this Validation in the Software Plan

```
(TRIAL1-VERIFY
 ((:execute trial1) (:on-entry :report) (:on-exit :report)
  ((:after setskips)
   ((:test (skiptable-correct word skiptable))
    (:on-pass :report) (:on-fail :report)))))
```

We see that we told it to report on the result of the test **skiptable-correct** in both cases of a pass or a failure of the test. In the case of our implementation of Find Word it reports a failure. This looks peculiar in light of the correct output, but was intentionally used in this example to illustrate that *working* software is not always *valid* OR correct!

Furthermore we now go on to show that by executing **TRIAL2** the output can be made wrong. **TRIAL2** is exactly the same as **TRIAL1** except for two extra spaces between "This" and "sentence" in the text:

```
Select and execute one of the plan validations:
 0 = Execute no validation
 1 = LOAD-FIND-WORD
 2 = TRIAL1
 3 = TRIAL1-VERIFY
 4 = TRIAL1-DEBUG
 5 = TRIAL2
 6 = LOAD-NEW-SETSKIPS
Enter a number from 0 to 6 -> 5

The word "sentence" occurs 2 times in the text
"This   sentence is an example sentence for finding the word sentence"
```

A serendipitous discovery here was that verification and debugging of programs are not only closely related but are part of a continuum, such that where verification ends blends into debugging and vice versa. If debugging is made sophisticated enough, at some point in the continuum debugging and verification are one and the same. Furthermore verification can support debugging and vice versa. We give a taste of this idea in this Find Word example. The validation **TRIAL1-DEBUG** is pretty much like **TRIAL1-VERIFY**

```
(TRIAL1-DEBUG
 ((:execute trial1) (:on-entry :report) (:on-exit :report)
```

```
((:after setskips)
.((:test (skiptable-correct word skiptable))
  (:on-fail :report
            ((:record (list word skiptable) skiptable-snapshot)))))))
```

What is different is that we take advantage of when the verification fails to do some debugging. So given :on-fail we not only report the results of the test but we record a snapshot of the skiptable.

The actual interaction with the ASP tool when selecting this Validation goes as follows

```
Select and execute one of the plan validations:
 0 = Execute no validation
 1 = LOAD-FIND-WORD
 2 = TRIAL1
 3 = TRIAL1-VERIFY
 4 = TRIAL1-DEBUG
 5 = TRIAL2
 6 = LOAD-NEW-SETSKIPS
Enter a number from 0 to 6 -> 4

------------------------------------------------------------------------------
| ASP controlling FIND-WORD plan.                                            |
| Entering: TRIAL1                                                           |
------------------------------------------------------------------------------

------------------------------------------------------------------------------
| ASP controlling FIND-WORD plan.                                            |
| Verification SKIPTABLE-CORRECT : AFTER point SETSKIPS.                     |
| Result = ***FAILED***                                                      |
------------------------------------------------------------------------------

The word "sentence" occurs 3 times in the text
"This sentence is an example sentence for finding the word sentence"

------------------------------------------------------------------------------
| ASP controlling FIND-WORD plan.                                            |
| Exiting: TRIAL1                                                            |
------------------------------------------------------------------------------
```

At this point the results look the same as the TRIAL1-VERIFY because we patterned TRIAL1-DEBUG after it. But since we included the :record given an :on-fail condition we can expect to see an attribute value for SETSKIPS, namely a snapshot of it when the test failed.

```
Select ASP activity for FIND-WORD plan:
```

```
 0 = Exit ASP tool.
 1 = Select a plan validation for execution.
 2 = View PO attributes.
 3 = Change plan defaults.
Enter a number from 0 to 3 -> 2

Select a Program Object to view its attributes:
 0 = Select nothing
 1 = SETSKIPS
Enter a number from 0 to 1 -> 1

Program object SETSKIPS has the following attributes

SKIPTABLE-SNAPSHOT is a SINGLE value =
("sentence"
 #(8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
   8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
   8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 1 8 -3 8 8 8 8 8 8 8 2 8 8 8
   8 8 4 8 8 8 8 8 8 8 8 8 8 8))
```

In the top level menu we choose `View PO attributes` (PO stands for Program Object), which shows that the program object `SETSKIPS` now has recorded attributes. When we look at the attributes we see an attribute called `SKIPTABLE-SNAPSHOT` which has the value of the word given to `SETSKIPS` and the actual contents of the table when the test failed. The table looks pretty good for the word "sentence" except for one thing. Since this is an ascii value indexed table we see a correct skip of 4 for the "t" position in the table, but just before it in the "s" position we see an 8 and expect to see a 7 since "s" in "sentence" is seven characters away from the end of the word "sentence". Aha! Could it be that we have a problem with the indexing of the word and especially of the first character? Yes indeed. When we translated the algorithm from Pascal in our text book to actual code we forgot that Pascal defaults to 1 origin arrays and in this case we want 0 origin arrays.

We correct this problem with a new implementation of `SETSKIPS` and now have ASP bring in this implementation with a Validation called `LOAD-NEW-SETSKIPS`.

```
Select and execute one of the plan validations:
 0 = Execute no validation
 1 = LOAD-FIND-WORD
 2 = TRIAL1
 3 = TRIAL1-VERIFY
 4 = TRIAL1-DEBUG
 5 = TRIAL2
 6 = LOAD-NEW-SETSKIPS
Enter a number from 0 to 6 -> 6

-------------------------------------------------------------------------
```

```
| ASP controlling FIND-WORD plan.                                          |
| Loaded: NEW-SETSKIPS-IMPLEMENTATION                                      |
---------------------------------------------------- ----------------------
```

And *now* we retry TRIAL2 and and it works:

```
Select and execute one of the plan validations:
 0 = Execute no validation
 1 = LOAD-FIND-WORD
 2 = TRIAL1
 3 = TRIAL1-VERIFY
 4 = TRIAL1-DEBUG
 5 = TRIAL2
 6 = LOAD-NEW-SETSKIPS
Enter a number from 0 to 6 -> 5

The word "sentence" occurs 3 times in the text
"This   sentence is an example sentence for finding the word sentence"
```

Finally, just as a detail, we can retry **TRIAL1-VERIFY** and see that it now passes the test:

```
Select and execute one of the plan validations:
 0 = Execute no validation
 1 = LOAD-FIND-WORD
 2 = TRIAL1
 3 = TRIAL1-VERIFY
 4 = TRIAL1-DEBUG
 5 = TRIAL2
 6 = LOAD-NEW-SETSKIPS
Enter a number from 0 to 6 -> 3

-------------------------------------------------------------------------------
| ASP controlling FIND-WORD plan.                                             |
| Entering: TRIAL1                                                            |
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
| ASP controlling FIND-WORD plan.                                             |
| Verification SKIPTABLE-CORRECT : AFTER point SETSKIPS.                      |
| Result = PASSED!                                                            |
-------------------------------------------------------------------------------

The word "sentence" occurs 3 times in the text
"This sentence is an example sentence for finding the word sentence"

-------------------------------------------------------------------------------
| ASP controlling FIND-WORD plan.                                             |
| Exiting: TRIAL1                                                             |
-------------------------------------------------------------------------------
```

A.3 Complete Semantics of the Software Plan

In this section the complete semantics of the Software Plan are explained by listing the keyword arguments of the function **specify-plan** and their meanings. Section A.6 contains the complete syntax of the Software Plan. Since keyword arguments are Common Lisp keywords, they can be specified in any order in relation to each other that the user finds most readable. Requiring quoting of these keyword arguments, such as a quoted list, was done intentionally, so that one could use variables or expressions for keyword arguments values if they needed that flexibility and also to make it easier to programmatically generate Software Plans. The term *plan scoped identifiers* refers to all of the symbols mentioned in the Software Plan that only have values within the scope of the Software Plan. When the software plan is interpreted by ASP *conditions* in the software plan cause ASP to arrange for the special handling of program objects in the implementation code such that when those objects are invoked randomly within the implementation, ASP will get control. We refer to this arrangement as *conditioned* code and the invocation of such objects as *triggering*.

For most of the keywords a brief description suffices. We first list and give brief descriptions of all the keywords. The keywords that need more detailed descriptions will be expounded later. We now list the **specify-plan** keywords:

:name - The name of the plan (such as **find-word**). This needs to be a symbol. This is used to distinguish one Software Plan from another and in report identification.

:log-file - A string representing the file specification of the file that the results of a **:log** Validation will get appended to. If not specified the log file defaults to **/.ASP-log**.

:verbose-loading - Set to **t** if you want the actual files being loaded displayed during a **:load** action.

:undefined-attribute-value - When referencing the values of **:record** attributes, if the attribute is undefined this value is returned. It defaults to the symbol **:undefined**.

:collection-coercion-type - When referencing the value of **:collect** attributes, which are recorded as sequences, they will be coerced to this data type. The default value is **list**.

:globals - A list of the global clauses that associates plan scoped identifiers with global identifiers of the host language.

:specifications - A list of specification clauses that associates file specification strings with plan scoped identifiers. These files will contain user defined code representing verifications.

:implementations - A list of implementation clauses that associates file specification strings with plan scoped identifiers. These files will essentially be the code of the user's software implementation, including multiple implementations.

:executables - A list of execution clauses that defines how ASP will pass control to the host language code.

:verification-points - A list of clauses that defines where in the users code verifications can take place.

:verifications - A list of clauses that defines how ASP will invoke user defined Verifications.

:sub-validations - A list of user defined validations. These are exactly the same as `:validations` but do not appear in the ASP tool selection menu.

:validations - A list of user defined validations.

A.3.1 Software Plan Constants

Constants, referred to as *plan constants*, may appear in the Software Plan, in certain positions. They may be either an integer or a string. For example

```
12345
-1
"Beginning of test"
```

are permissible plan constants.

A.3.2 Plan Scoped Identifiers

Several constructs in a Software Plan take arguments much like a function call. But one construct, the `:verifications-points` clauses *give* rather than take arguments. This means that when a plan scoped identifier is included in the clause, its value is not passed to the function named in the clause but instead the plan scoped identifier is bound to the value of the actual parameter when the function gets triggered. In the Find Word example the `:verifications-points` clause

```
(setskips word :output skiptable)
```

causes the plan scoped identifier **word** to get bound to the **setskips** function's first argument value whenever **setskips** happens to be triggered by the implementation code. Similarly the plan scoped identifier **skiptable** gets bound to the returned value of **setskips**. Note that such bindings are only ready to be made in the scope of a **:before** or **:after** condition. Once such a binding actually occurs by a triggering, its scope in the Software Plan is indefinite. Or, in other words, the scoped identifier will have that value in the local scope of the plan indefinitely until the next time it is caused to be changed as specified in the plan. So in the Find Word example once **word** gets bound to **setskip**'s first argument value that will be its visible value in the Software Plan until the next triggering of **setskips**.

All other occurrences of plan scoped identifiers pass the value bound to the identifier to a specified function in the enclosing construct. In the Find Word example considering the construct

```
(skiptable-correct word skiptable)
```

the verification (function) **skiptable-correct** gets passed the current plan bound values for the plan scoped identifiers **word** and **skiptable**.

Plan scoped identifiers are also used to specify attributes and the attribute's value binding has the same scope rules as a plan scoped identifier's regular value binding. In the Find Word example considering the construct

```
(:record (list word skiptable) skiptable-snapshot)
```

the plan scoped identifier **skiptable-snapshot** becomes the attribute that gets a value as a result of the **:record** action. A plan scoped identifier can be both a regular value and an attribute value at the same time.

A.3.3 Software Plan Arguments

As mentioned in the previous section various constructs can take arguments. Following sections will describe the meaning of the arguments for the particular construct. Unless otherwise stated an argument specified in the Software Plan, with the exception of arguments for verification points, can be one of the following:

1. A plan constant (as defined above)
2. A plan scoped identifier (as defined above)
3. A recorded attribute value reference
4. A verification form (as defined below)

:implementations - A list of implementation clauses that associates file specification strings with plan scoped identifiers. These files will essentially be the code of the user's software implementation, including multiple implementations.

:executables - A list of execution clauses that defines how ASP will pass control to the host language code.

:verification-points - A list of clauses that defines where in the users code verifications can take place.

:verifications - A list of clauses that defines how ASP will invoke user defined Verifications.

:sub-validations - A list of user defined validations. These are exactly the same as `:validations` but do not appear in the ASP tool selection menu.

:validations - A list of user defined validations.

A.3.1 Software Plan Constants

Constants, referred to as *plan constants*, may appear in the Software Plan, in certain positions. They may be either an integer or a string. For example

```
12345
-1
"Beginning of test"
```

are permissible plan constants.

A.3.2 Plan Scoped Identifiers

Several constructs in a Software Plan take arguments much like a function call. But one construct, the `:verifications-points` clauses *give* rather than take arguments. This means that when a plan scoped identifier is included in the clause, its value is not passed to the function named in the clause but instead the plan scoped identifier is bound to the value of the actual parameter when the function gets triggered. In the Find Word example the `:verifications-points` clause

```
(setskips word :output skiptable)
```

In the case of a Software Plan constant the constant's value is used as the actual argument. In the case of the plan scoped identifier its Software Plan currently bound value is used as the actual argument.

A recorded attribute value has the form:

(*object-name attribute-identifier*)

where *object-name* is the name of the object that gets the attribute, *attribute-identifier* is the plan scoped identifier that names the attribute. The value of the attribute *attribute-identifier* of object named *object-name* is used as the actual argument.

Verification forms are defined in the section **Software Plan** `:verifications`. The value returned as a result of ASP invoking the verification is used as the actual argument.

A.3.4 Software Plan `:globals`

The `:globals` value is a list of clauses of the form:

(*plan-arg global-val*)

the *global-val* can be either a plan constant or a symbol in which case it is interpreted as an identifier of the host language that names a global variable. *plan-arg* is a plan scoped identifier that is associated with the *global-val* in the scope of the whole Software Plan. For example, in

```
:globals '((pdb *person-data-base*)
           (alert "We are changing implementations at this point"))
```

`pdb` is a plan scoped identifier that is associated with the global variable `*person-data-base*` and `alert` is a plan scoped identifier that is associated with the constant string `"We are changing implementations at this point"`.

A.3.5 Software Plan `:specifications`

The `:specifications` value is a list of clauses of the form

(*specification-name string string ...*)

specification-name is a plan scoped identifier that names an implementation. The implementation is associated with *string string ...* that are file specifications of files in the host operating system environment. For example,

```
:specifications
'((SKIP-TABLE-SPECIFICATION
    "/systems/asp/examples/find-word/verify"))
```

names one specification called **SKIP-TABLE-SPECIFICATION** that gets associated with one file in the host operating system.

A.3.6 Software Plan :implementations

The **:implementations** value is a list of clauses of the form

(implementation-name string string ...)

implementation-name is a plan scoped identifier that names an implementation. The implementation is associated with *string string ...* that are file specifications of files in the host operating system environment. For example

```
:implementations
'((FIND-WORD-SUBLINEAR-IMPLEMENTATION
    "/systems/asp/examples/find-word/sublinear"
    "/systems/asp/examples/find-word/trials"))
```

names one implementation called **FIND-WORD-SUBLINEAR-IMPLEMENTATION** that gets associated with two files in the host operating system.

A.3.7 Software Plan :executables

The **:executables** value is a list of clauses of the form

(executable-name arg arg ...)

where *executable-name* is the name of a function in the software implementation and *arg arg ...* are plan scoped identifiers. The *executable-name* is usually named in the *:execute* action which results in a call to the implementation's function. In this call the actual values of *arg arg ...* are passed as parameters.

A.3.8 Software Plan :verification-points

The :**verification-points** value is a list of clauses of the form

(*verification-point arg arg ...* :**output** *output-arg output-arg ...*)

where *verification-point* is the name of a function in the software implementation and *arg arg ...* are plan scoped identifiers that get bound to the function's actual parameters when the function gets triggered. If :**output** is specified then the plan scoped identifiers *output-arg output-arg ...* will get bound to the function's output parameters.

The *verification-point* also serves as an identifier for the :**before**, :**after** conditions. The above bindings will only occur during the scope of a :**before** or :**after** condition and during that condition when the verification point gets triggered.

A.3.9 Software Plan :verifications

The :**verifications** value is a list of clauses of the form

(*verification-name arg arg ...*)

Where *verification-name* is the name of a verification that the user writes. The verification must be a predicate function. The *arg arg ...* are plan scoped identifiers whose values get passed to the verification whenever the verification is invoked in a Software Plan validation.

You may specify *arg arg ...* in the verification clause mentioned above or in a verification *form* that occurs as an argument in a validation or both. The arguments in the verification form will always override the arguments in the verification clause. Some modifications to the Find Word example will make this clear. In the Find Word example we specified:

```
:verifications '((skiptable-correct) ...)

(TRIAL1-VERIFY ...
  ... (:test (skiptable-correct word skiptable)))
```

But we could have gotten exactly the same effect with

```
:verifications '((skiptable-correct word skiptable) ...)

(TRIAL1-VERIFY ...
  ... (:test skiptable-correct))
```

Furthermore if we had specified

```
:verifications '((skiptable-correct word skiptable) ...)

(TRIAL1-VERIFY ...
  ... (:test (skiptable-correct new-word new-skiptable)))
```

Then the **skiptable-correct** verification would have received parameters that were the values of the plan scoped identifiers **new-word** and **new-skiptable** since the identifiers in the verification form override the identifiers in the verification's clause.

In general a verification form looks like

verification-name
(*verification-name arg arg ...*)

where *verification-name* is defined in a verification clause in the :**verifications** list. In the first form ASP looks at the verification clause to compute the verification's actual arguments. In the second form *arg arg ...* is used.

A.3.10 Software Plan :sub-validations

The semantics of :**sub-validations** are exactly the same as the semantics of :**validations**. The only difference is that they do not show up in the validations selection menu in the ASP tool. This is handy for validations that you want to invoke from other validations, like subroutines, but not be selectable.

A.3.11 Software Plan :validations

The :**validations** value is a list of clauses of the form

(*validation-name action action ...*)

where *validation-name* appears in the ASP tool validations selection menu. *validation-name* may also occur any place that an action can occur in any other validation. The effect of selecting *validation-name* in the ASP tool menu or invoking it from another validation is simply to have the ASP tool interpret the actions *action action ...* . The following sections will explain these actions in more detail.

A.3.11.1 Action Condition Semantics

The general schema of any ASP Software Plan validation is based on actions and conditions which have the following forms:

(*action condition condition ...*)
(*condition action action ...*)

What this means is that given *action* ASP will perform that action on the implementation code subject to that code being **conditioned** by *condition condition ...* . And given the one instance of **conditioned** code *condition*, ASP will perform the actions *action action ...* on the implementation code.

Furthermore any *action* in *action action ...* has precisely the form of the top form and any *condition* in *condition condition ...* has precisely the form of the bottom form. This recursive definition implies that actions can have conditions which can have actions which can have conditions ...etc. This gives the Plan writer the capability to have nested actions based on verification conditions carried out to finer and finer levels of detail. This is a natural paradigm for verifying software.

One should not assume that ASP performs this chain of actions and conditions in a sequential control thread as in a conventional programming language. Consider the following construct:

```
(action-a ((:before verification-point-1) action-1)
          ((:before verification-point-2) action-2)
          ((:before verification-point-3) action-3 action-4))
```

`action-2` could occur any number of times before `action-1` depending on when `verification-point-2` and `verification-point-1` trigger in the implementation code. In fact, `action-1` might never occur if `verification-point-1` never triggers. However, if `verification-point-3` triggers, `action-3` and `action-4` are guaranteed to occur in succession.

Within any validation the only kinds of top level entities that can occur are:

Predefined actions - actions that are predefined by ASP

Predefined conditions - conditions that are predefined by ASP

Predefined validations - ASP canned validations which can occur anywhere that an action can occur

User defined validations - User defined validations whose names can occur anywhere that an action can occur

The predefined actions conditions and validations are explained and itemized in the following sections.

A.3.11.2 Predefined Actions

The predefined actions have the following meanings:

:load - Specified specifications or implementations are loaded into the ASP conditioned host implementation.

:execute - A specified executable is control executed by ASP. Usually it will have conditions that affect verifications.

:engage - Intended to be the same semantics as **:execute** but provides an escape mechanism for user intervention or manual code execution.

:test - Specified verifications that act as predicates are control executed by ASP. The results of these Verifications will each respond to subsequent **:on-pass** and **:on-fail** conditions.

:record - The value of the specified verification is recorded as the value of the specified object's attribute.

:collect - The same meaning as **:record** except that the value is recorded in a sequence which becomes the attribute's value. The sequence is ordered by most recently recorded first.

:report - There is a predefined **:report** validation that does a specific canned thing depending on its enclosing condition. The **:report** action gives the user control over what is reported and when it is reported.

:log - There is a predefined **:log** validation that does a specific canned thing depending on its enclosing condition. The **:log** action gives the user control over what is logged and when it is logged.

The form of the **:load** action is

(**:load** *name name* ...)

where *name name* ... are the names of implementations or specifications defined by the **:implementations** or **:specifications** keywords. If the existing host implementation is conditioned, ASP maintains the conditions. Multiple implementations can be introduced this way by loading and overlaying same named functions.

The form of the **:execute** action is

(**:execute** *name*)

where *name* is an executable defined by the **:executables** keyword. When interpreted, ASP will first condition the implementation based on the conditions of the **:execute** then call the host function named by the executable.

The form of the **:engage** action is

```
(:engage :listener)
```

where **:listener** is the type of engagement. Others types of engagement may be introduced in future ASP releases. When interpreted ASP will first condition the implementation based on the conditions of the **:engage** then give control to the ASP Lisp Listener. While in the ASP Lisp Listener by evaluating a **:c** the user may give control back to the ASP tool. The basic idea of **:engage** is to have the same semantics as **:execute** but without the execution of any user code. This has two uses. First, as a way for the user to inspect his environment at some point in the Software Plan. And second as a way for the user to manually execute code at some point in the Software Plan while his code is conditioned by the Software Plan. Since this is a difficult concept to convey, an example of doing this by extending the Find Word example is given in a following section.

The form of the **:test** action is

```
(:test verification-form verification-form ...)
```

where *verification-form* is a verification form as defined above. If the **:test** action has any **:on-pass** or **:on-fail** conditions, ASP will invoke the verification predicate functions based on *verification-form verification-form* ..., perform the **:on-pass** actions for **every** predicate that is true, and perform the **:on-fail** actions for every predicate that is false. The user should note that given the following:

```
((:test v1 v2 v3) (:on-pass action1))
```

action1 will be performed for every predicate of **v1** through **v3** that is true. If this is not desired then something like

```
((:test v1) (:on-pass action1))
((:test v2) (:on-pass action2))
((:test v3) (:on-pass action3))
```

should be written.

The **:record** action records a plan scoped attribute value. It has one of two forms:

(:**record** *val-arg object attribute*)
(:**record** *val-arg attribute*)

val-arg is a Software Plan argument as defined above. *object* and *attribute* are plan scoped identifiers. As a result of the :**record** action the object named by *object* will acquire an attribute named by *attribute* with a value determined by the actual value of *val-arg*. This attribute can then be later viewed via an ASP menu or used in other places in the validation or other validations.

The only difference in the second form is that *object* does not appear. In this case ASP uses the name of the verification point in the last enclosing :**before** or :**after** clause as the *object*.

The :**collect** action has exactly the same form and semantics as the :**record** action except that the attribute value is a sequence and the computed :**record** value is made the first element of the sequence. When such a *collected* attribute is used as a Software Plan argument the actual value is computed by coercing the sequence to the data type specified by the Software Plan keyword :**collection-coercion-type**. If not specified this keyword defaults to the **list** data type.

The :**report** and :**log** actions are similar in what the :**report** and :**log** predefined validations do. The predefined validations do a specific canned thing depending on their enclosing condition, where as the actions do a user specified thing where ever they are specificed in the Software Plan. Their form is

(:**report** *format-string val-arg1 val-arg2 ...*)
(:**log** *format-string val-arg1 val-arg2 ...*)

where *format-string* is a Common Lisp type format control string and *val-arg1 val-arg2 ...* are 0 or more Software Plan arguments as defined above. The resulting values of these arguments are consumed exactly as the arguments would be consumed by the Common Lisp format control string. An example of using :**report** with the Find Word example is given in a following section.

A.3.11.3 Predefined Conditions

The predefined conditions have the following meanings:

:on-entry :on-exit - given an action, before the action is performed the actions of the :**on-entry** condition are performed. :**on-exit** is similar but applies after the action is performed.

:on-pass :on-fail - The results of the last previous :**test** verification trigger these conditions. If the result of the verification predicate is *true* the :**on-pass** condition's actions are performed. If the result of the verification predicate is *false* the :**on-fail** condition's actions are performed.

:before :after - If a previously specified **:execution** causes the triggering of the **:before** or **:after** condition's specified verification point, the condition's actions are performed before the verification point in the case of a **:before** condition and after the verification point in the case of an **:after** condition.

A.3.11.4 Permissible Conditions

All conditions can have any actions or validations. But not all conditions that actions can have, have meaning. Only those that have meaning will be performed. The ones that have meaning are as follows:

Action	Meaningful Conditions
:execute	:before :after :on-entry :on-exit
:engage	:before :after :on-entry :on-exit
:test	:on-pass :on-fail
:load	:on-entry :on-exit
:record	none
:collect	none
:report	none
:log	none

A.3.11.5 Predefined Validations

The predefined validations are as follows:

:report - The results of the last action of the enclosing condition are displayed to the user in some way.

:log - The results of the last action of the enclosing condition are logged to the log file specified in the Software Plan.

:abort - Aborts all nested validations up to the current top level validation.

A.3.11.6 Using Validations

The form of any validation is simply

validation

where *validation* is the name of a predefined or a user defined validation. This validation form *validation* can appear anywhere that an action can appear. For example we could write

```
(:on-fail :report DO-MY-VALIDATION :abort)
```

A.4 More Find Word Examples

In this section we use modifications of the Find Word example to illustrate some of ASP's capabilities that were not illustrated by the Find Word program example itself.

A.4.1 An Example Using the :report and :log Actions

In the Find Word example we had a validation called TRIAL1-DEBUG

```
(TRIAL1-DEBUG
  ((:execute trial1) (:on-entry :report) (:on-exit :report)
   ((:after setskips)
    ((:test (skiptable-correct word skiptable))
     (:on-fail :report
               ((:record (list word skiptable) skiptable-snapshot)))))))
```

After selecting this validation with the ASP tool we then later manually looked at the **skiptable-snapshot** attribute value to visualize what was going on. By using the :report action we could have specified that we want to see the attribute value during the validation.

```
(TRIAL1-DEBUG
  ((:execute trial1) (:on-entry :report) (:on-exit :report)
   ((:after setskips)
    ((:test (skiptable-correct word skiptable))
     (:on-fail :report
               ((:record (list word skiptable) skiptable-snapshot))
               ((:report "~%Word and Skiptable =~%~a~%"
                         (setskips skiptable-snapshot)))))))))
```

We could get the same effect without having the plan execute **trial1** but instead put us into the ASP Lisp Listener from where we could execute **trial1** and **trial2** or any number of manual executions or inspections. When we are though with these manual activities we enter **:c** and the ASP Lisp Listener puts us back in the ASP tool where we started. We could add a validation called simply **TRIAL-VERIFY** to accomplish this.

```
(TRIAL-VERIFY
  ((:engage :listener) (:on-entry :report) (:on-exit :report)
   ((:after setskips)
    ((:test (skiptable-correct word skiptable))
     (:on-pass :report) (:on-fail :report)))))
```

Then upon selecting the **TRIAL-VERIFY** validation in the ASP tool we could get the following dialog:

```
------------------------------------------------------------------------
| ASP controlling FIND-WORD plan.                                      |
| Entering: ASP Listerer                                               |
------------------------------------------------------------------------


ASP Listener with validation TRIAL-VERIFY engaged.
 Enter :c to continue with ASP tool.

ASP> (trial1)

------------------------------------------------------------------------
| ASP controlling FIND-WORD plan.                                      |
| Verification SKIPTABLE-CORRECT : AFTER point SETSKIPS.               |
| Result = ***FAILED***                                                |
------------------------------------------------------------------------

The word "sentence" occurs 3 times in the text
"This sentence is an example sentence for finding the word sentence"
NIL
ASP> (trial2)

------------------------------------------------------------------------
| ASP controlling FIND-WORD plan.                                      |
| Verification SKIPTABLE-CORRECT : AFTER point SETSKIPS.               |
| Result = ***FAILED***                                                |
------------------------------------------------------------------------

The word "sentence" occurs 2 times in the text
"This   sentence is an example sentence for finding the word sentence"
```

```
NIL
ASP> :c

------------------------------------------------------------------------
| ASP controlling FIND-WORD plan.                                      |
| Exiting: ASP Listener                                                |
------------------------------------------------------------------------


Select ASP activity for FIND-WORD plan:
 0 = Exit ASP tool.
 1 = Select a plan validation for execution.
 2 = View PO attributes.
 3 = Change plan defaults.
Enter a number from 0 to 3 -> 0
ASP::DONE
```

While in the ASP Lisp Listener we evaluated the expression (`trial1`). Notice that when we did this the effect was the same as selecting `TRIAL1-VERIFY` except that we returned back to the ASP Lisp Listener at which point we evaluated the expression (`trial2`).

The ASP Lisp Listener is the same as an ordinary Lisp Listener except that the prompt is `ASP>` and when we evaluate `:c`, which stands for continue, we are returned to the ASP tool.

Another use for the `:engage` action would be to include it without any conditions within a nested set of actions and conditions so that the user could manually inspect his environment in that state. For example given

```
(action1 (condition1 action2 action3
                     (action4 (condition2 action5
                                          ((:engage :listener))))))
```

In the case of `condition2`, `action5` would take place followed by the ASP Lisp Listener getting control.

A.5 Using ASP with Specification Languages

Using ASP with a specification language further brings into focus its capabilities. Formal specifications play the part of executable specifications, testable verifications are short logic expressions, decomposition of specifications maps into the idea of partial specifications, the before and after effect of complex transformations is aligned with the idea of centralizing multi-point tests in validations, and one has more control over generating multiple implementations.

We demonstrate some of these capabilities by giving an example of using ASP with the Refine specification language as the host language. At the same time we demonstrate more features of the Software Plan. We use an example called Buses which is a toy example of resource scheduling and planning. We will show an early software evolution phase Software Plan that discovers a constraint fault in the knowledge-base by using decomposition of specifications. We will not emphasize the interaction with the ASP tool as much as we did with the Find Word example. The complete sources of the Buses example are delivered with the ASP software. To follow this example more thoroughly we hope that the user will load the Buses software implementation and the Buses Software Plan and apply the ASP tool by stepping through the validations in the Software Plan.

A.5.1 The Buses Example

The Buses example is one of scheduling resources for a bus system. These resources are buses, drivers, routes and trips which are modeled in a knowledge base using the Refine language. Constraints such as what types of buses can be used on each route, what routes a driver is qualified to drive, and limits on the amount of time a driver can drive during a day can be expressed as assertions using logic and set-theoretic constructs.

We build a Buses implementation that will create a schedule and refine the schedule into a sequence of events. We build executable specifications by writing verifications that are logic expressions in Refine. These verifications when taken as a whole validate the constraints mentioned above. In the example we invalidate the constraints by introducing a fault in them. A Software Plan validation detects this by failing one of the composite tests. We then try a validation that decomposes that particular test into partial specifications and discover the fault.

A.5.2 The Buses Implementation

We only show the top level functions for the buses implementation here. The BUSES example is an extension of a Bus Scheduling example given by Reasoning Systems in their Refine tutorial. If the user is interested, all of the code for the Buses example

is delivered with the ASP software. The function **GENERATE-SCHEDULES** generates all possible schedules. It does most of its work by calling **RECURSIVE-CREATE-SCHEDULE**.

```
"Top-level function that generates all possible schedules and returns
 the number found."
function GENERATE-SCHEDULES (b-w: bus-world) : integer
  =  initialize-bus-world (b-w);
     initialize-globals();
     recursive-create-schedule (b-w);
     report-scheduling-done (b-w);
     *schedule-count*

"Recursive scheduling function for the bus world.  Incorporates
 backtracking and finds all schedules."
function RECURSIVE-CREATE-SCHEDULE (b-w) : bus-world
 =  let ( rts = the-routes (b-w))
     let ( r = Route-with-Earliest-Uncovered-Time (rts))
      let ( return-time = Last-bus-trip-return-time (r))
         Report-new-recursion-level-and-route-data (r, return-time);
         (if return-time >= end-time-requirement(b-w)
             then report-schedule-found (rts)
             else
             %% Generate all legal combinations of
             %% bus & driver for next bus-trip.
             let (legal-bus-driver-pairs =
                   generate-legal-bus-driver-pairs (b-w, r, return-time))
               (if empty (legal-bus-driver-pairs)
                   then report-schedule-failure (r)
                   else
                   (enumerate
                     b-d:tuple( bus, driver) over legal-bus-driver-pairs
                     do
                     let (new-bus-trip = make-structure
                                         ('the-bus-trip @(newsymbol('TR))
                                          with-bus-trip-driver @(b-d.2)
                                          with-bus-trip-bus @(b-d.1)
                                          on-route @r
                                          starting-at @return-time' ))
                       report-new-bus-trip (new-bus-trip);
                       Recursive-Create-schedule (b-w);
                       Retract-bus-trip (new-bus-trip))));
         Report-backtracking();
         b-w
```

A.5.3 Buses Executable Specifications

The following verifications serve as executable specifications for the Buses example. The natural language specification that the executable specification verifies precedes

each test verification. Each verification is based on resource constraints in the Buses object world. Notice that all of the verifications are expressed in Refine as logic formulas. Also notice that they are all quantified conjunctions. This helps us when we want to decompose specifications into partial specifications.

```
"TEST-P-1: A partially completed schedule has a schedule for every route."
function TEST-P-1 (world: bus-world):boolean  =
  fa(r) (r in the-routes(world) =>  ex(s)
    (s = route-schedule(r)
    % the times in s make s a continuous schedule starting at 0.
   & ((defined?(last(s)) & defined?(bus-trip-start(last(s))))
       => bus-trip-start(last(s)) = 0.0)
   & fa(t1)
      (t1 in s =>
        ((defined?(bus-trip-end(t1)) and defined?(bus-trip-start(t1)))
          => (bus-trip-end(t1) - bus-trip-start(t1)
                = trip-time(bus-trip-bus(t1),r))))
   & fa(t1, t2)
       ((t1 in s & t2 in s) =>
         (s = [..,t2,t1,..] => bus-trip-start(t2) = bus-trip-end(t1)))))

"TEST-C-1: A completed schedule has a schedule for every route."
function TEST-C-1 (world: bus-world):boolean  =
  fa(r) (r in the-routes(world) => ex(s)
    (s = route-schedule(r)
        %       the times in s make s a continuous schedule covering the
        %       whole time period.
  & bus-trip-start(last(s)) = 0.0
  & bus-trip-end(first(s)) >= end-time-requirement(world)
  & fa(t1)
      (t1 in s =>
         bus-trip-end(t1) - bus-trip-start(t1) = trip-time(bus-trip-bus(t1),r)
  & fa(t1, t2)
      ((t1 in s & t2 in s) =>
        (s = [..,t2,t1,..] => bus-trip-start(t2) = bus-trip-end(t1)))))


"TEST-2: No two buses are in use at the same time,
and there is at least 15 minutes for refueling between uses."
function TEST-2 (world: bus-world):boolean  =
  fa(r) (r in the-routes(world) =>  ex(s)
    (s = route-schedule(r) &
      fa(t1,t2) ((t1 in s & t2 in s) =>
       (( defined?(bus-trip-start(t1)) & defined?(bus-trip-end(t1)) &
          defined?(bus-trip-start(t2)) & defined?(bus-trip-end(t2))) =>
        (t1 ~= t2 =>
          ((  bus-trip-start(t1) < bus-trip-start(t2)
            & bus-trip-end(t1) <= bus-trip-start(t2))   or
           (  bus-trip-start(t2) < bus-trip-start(t1)
            & bus-trip-end(t2) <= bus-trip-start(t1)))))))))
```

```
  &
  let (schedules = { route-schedule(r) | (r) r in the-routes(world)})
    fa(s1,s2) ( (s1 in schedules  &  s2 in schedules) =>
      ( s1 ~= s2 =>
     fa(t1,t2) ( (t1 in s1  & t2 in s2) =>
       ( t1 ~= t2 =>
         (( defined?(bus-trip-start(t1)) & defined?(bus-trip-end(t1)) &
            defined?(bus-trip-start(t2)) & defined?(bus-trip-end(t2)) &
            defined?(bus-trip-bus(t1)) & defined?(bus-trip-bus(t2)) )  =>
               (bus-trip-bus(t1) = bus-trip-bus(t2)  =>
                     (bus-trip-end(t1) + 0.25 < bus-trip-start(t2)  or
                      bus-trip-end(t2) + 0.25 < bus-trip-start(t1)))))))))

"TEST-3: No two drivers are on different trips at the same time,
 and they are not driving more that 8 hours a day."
function TEST-3 (world: bus-world):boolean  =
  fa(d) (d in the-drivers(world) =>
    total-driving-time(d) <= 8.0)
  &
  fa(r) (r in the-routes(world) =>  ex(s)
    (s = route-schedule(r) &
      fa(t1,t2) ((t1 in s & t2 in s) =>
        ( t1 ~= t2  =>
           ( (  bus-trip-driver(t1) = bus-trip-driver(t2) ) =>
              ((bus-trip-start(t1) > bus-trip-end(t2)  or
                bus-trip-end(t1) <= bus-trip-start(t2)))))))))

"TEST-4:  All the other problem-specific constraints."
function TEST-4 (world: bus-world):boolean  =
  fa(b) (b in the-buses(world) =>
  fa(bt) (bt in bus-bus-trips(b) =>
    bus-trip-driver(bt) in drivers-trained-for(bus-trip-route(bt))
  & bus-size(b) in qualified-for(bus-trip-driver(bt))
  & yard-of-driver(bus-trip-driver(bt)) = bus-yard(bus-trip-bus(bt))
  & mountain-view-restriction?(b, bus-trip-route(b))
  & fremont-restriction?(b, bus-trip-route(b))))
```

A.5.4 Buses Constraint Fault

In the Software Plan the validation that introduces the constraint fault loads an implementation with the fault. This fault is in the constraint called BUS-DRIVER-CONSISTENCY which is a conjunction of three sub-constraints given a particular driver, bus and route:

1. The yard of the driver should be the same as the yard of the bus.
2. The driver should be qualified for the size of the bus.
3. The total hour limit is correct for the combination of driver, bus and route.

```
function BUS-DRIVER-CONSISTENCY
   (d:driver, b:bus, r:route) : boolean
 =
%%%  bus-yard (b) = yard-of-driver (d)  &
     bus-size (b) in qualified-for (d) &
     total-hour-limit-ok? (d, b, r)
```

For demonstration purposes we artificially comment out sub-constraint number 1. This is equivalent to leaving it out at some phase of the software evolution.

A.5.5 Buses Partial Executable Specifications

Because of the fault, TEST-4 will fail, so we decompose TEST-4 into partial specifications and create verifications TEST-4-1 through TEST-4-5:

```
function TEST-4-1 (world: bus-world):boolean  =
  fa(b) (b in the-buses(world) =>
  fa(bt) (bt in bus-bus-trips(b) =>
    bus-trip-driver(bt) in drivers-trained-for(bus-trip-route(bt))))

function TEST-4-2 (world: bus-world):boolean  =
  fa(b) (b in the-buses(world) =>
  fa(bt) (bt in bus-bus-trips(b) =>
    bus-size(b) in qualified-for(bus-trip-driver(bt))))

function TEST-4-3 (world: bus-world):boolean  =
  fa(b) (b in the-buses(world) =>
  fa(bt) (bt in bus-bus-trips(b) =>
    yard-of-driver(bus-trip-driver(bt)) = bus-yard(bus-trip-bus(bt))))

function TEST-4-4 (world: bus-world):boolean  =
  fa(b) (b in the-buses(world) =>
  fa(bt) (bt in bus-bus-trips(b) =>
    mountain-view-restriction?(b, bus-trip-route(b))))

function TEST-4-5 (world: bus-world):boolean  =
  fa(b) (b in the-buses(world) =>
  fa(bt) (bt in bus-bus-trips(b) =>
   fremont-restriction?(b, bus-trip-route(b))))
```

A.5.6 Buses Software Plan

We now show the Buses Software Plan that refers to the implementations and specifications that we discussed above.

```
(asp:specify-plan
 :name 'buses
 :specifications
 '((buses-specification "/systems/asp/examples/buses/vnv-spec")
   (buses-finer-specification "/systems/asp/examples/buses/vnv-spec-sub"))
 :implementations
 '((buses-base-implementation "/systems/asp/examples/buses/load-base"
                              "/systems/asp/examples/buses/imp1")
   (buses-faulty-implementation "/systems/asp/examples/buses/plant-bug"))
 :globals '((w2 *world2*))
 :executables '((generate-schedules w2))
 :verification-points '((report-new-bus-trip) (report-schedule-found))
 :verifications '((test-p-1 w2) (test-c-1 w2)
                  (test-2 w2) (test-3 w2) (test-4 w2)
                  (test-4-1 w2) (test-4-2 w2) (test-4-3 w2) (test-4-4 w2)
                  (test-4-5 w2) (1+) (>))
 :sub-validations
 '((SHOW-NO-REPORT-SCHEDULE-FOUND
    ((:record 0 report-schedule-found count)))
   (LIMIT-REPORT-SCHEDULE-FOUND
    ((:record (1+ (report-schedule-found count)) report-schedule-found count))
    ((:test (> (report-schedule-found count) 4)) (:on-pass :abort))))
 :validations
 '((LOAD-BUSES
    ((:load buses-base-implementation buses-specification)
     (:on-entry :report) (:on-exit :report)))
   (ORDINARY-RUN
    ((:execute generate-schedules)))
   (ORDINARY-TEST-RUN
    SHOW-NO-REPORT-SCHEDULE-FOUND
    ((:execute generate-schedules) (:on-entry :report) (:on-exit :report)
     ((:before report-new-bus-trip)
      ((:test test-p-1 test-2 test-3 test-4)
       (:on-pass :report) (:on-fail :report :abort)))
     ((:before report-schedule-found)
      LIMIT-REPORT-SCHEDULE-FOUND
      ((:test test-c-1 test-2 test-3 test-4)
       (:on-pass :report) (:on-fail :report :abort)))))
   (ORDINARY-TEST-RUN-WITH-SUSPECT
    ((:load buses-faulty-implementation)
     (:on-entry :report) (:on-exit :report))
    ORDINARY-TEST-RUN)
   (FIND-SUSPECT-TEST-RUN
    ((:load buses-faulty-implementation buses-finer-specification))
    ((:execute generate-schedules)
     ((:before report-new-bus-trip)
      ((:test test-p-1 test-2 test-3) (:on-fail :report))
      ((:test test-4)
       (:on-pass :report)
       (:on-fail :report)
       (:on-fail ((:test test-4-1 test-4-2 test-4-3 test-4-4)
```

```
              (:on-pass :report) (:on-fail :report :abort))))))
 ((:before report-schedule-found)
  ((:test test-c-1 test-2 test-3) (:on-fail :report))
  ((:test test-4)
   (:on-pass :report)
   (:on-fail :report
             ((:test test-4-1 test-4-2 test-4-3 test-4-4)
              (:on-pass :report) (:on-fail :report :abort)))))))
```

At this phase of the software evolution we have five validations:

1. LOAD-BUSES
2. ORDINARY-RUN
3. ORDINARY-TEST-RUN
4. ORDINARY-TEST-RUN-WITH-SUSPECT
5. FIND-SUSPECT-TEST-RUN

LOAD-BUSES simply loads the base implementation.

ORDINARY-RUN simply *executes* GENERATE-SCHEDULES. ASP will pass it the value of the plan scoped identifier **w2** which is associated in the Software Plan with the refine variable ***world2*** which points to the Buses world knowledge-base. By selecting this validation with the ASP tool you will see the output that GENERATE-SCHEDULES produces without any conditioning and output by ASP.

ORDINARY-TEST-RUN does the same thing as ORDINARY-RUN except that it conditions the implementation code before the verification point **report-new-bus-trip** by testing the verifications **test-p-1**, **test-2**, **test-3**, and **test-4**. If any one of these tests pass it will simply report that it passed, if any one fails it will report that it fails and abort the validation ORDINARY-TEST-RUN. Simultaneously it conditions the code before **report-schedule-found**. Since it was discovered that **report-schedule-found** triggers a huge number of times, ORDINARY-TEST-RUN invokes the validation LIMIT-REPORT-SCHEDULE-FOUND which limits the number of triggers to 5 then aborts.

When you select ORDINARY-TEST-RUN you will see that all tests pass. So we select ORDINARY-TEST-RUN-WITH-SUSPECT. Notice that this validation simply loads the constraint fault implementation that we discussed above and performs the ORDINARY-TEST -RUN validation. This is common; *we want to test the code the same way we did before, but with a different implementation of the same function.* Now with ORDINARY-TEST-RUN-WITH-SUSPECT we will see failures being reported and in particular **test-4** which will cause ASP to output:

```
-------------------------------------------------------------------------
| ASP controlling BUSES plan.                                           |
| Verification TEST-4 : BEFORE point REPORT-NEW-BUS-TRIP.               |
| Result = ***FAILED***                                                 |
-------------------------------------------------------------------------
```

Finally we select **FIND-SUSPECT-TEST-RUN** which will make use of our partial specifications we discussed above by loading the specification **buses-finer-specification**. In this validation we see a bit more complex nested testing going on. Given that **test-4** fails, ASP is directed to test the partial executable specifications of **test-4** namely **test-4-1** through **test-4-4**. This eventually results in a triggering that produces the ASP output:

```
-------------------------------------------------------------------------
| ASP controlling BUSES plan.                                           |
| Verification TEST-4-3 : BEFORE point REPORT-NEW-BUS-TRIP.             |
| Result = ***FAILED***                                                 |
-------------------------------------------------------------------------

-------------------------------------------------------------------------
| ASP controlling BUSES plan.                                           |
| Controlled aborting of validation                                     |
| FIND-SUSPECT-TEST-RUN ...                                             |
-------------------------- ----------------------------------------------
```

These results pin down the constraint fault and end this phase of software evolution. The current validations and verifications will be used over and over again for future phases.

A.6 Software Plan Complete Syntax

The syntax for an ASP Software Plan is expressed here in BNF. The usual BNF notations are used: The production symbol is ::=, angle brackets <> describe non-terminals, a bar | indicates alternatives, curly brackets followed by an asterisk {}* indicates 0 or more occurrences of the construct inside of the curly brackets, and anything else is a terminal symbol.

The Production tree starts at that described under **TOP LEVEL** and continues with that described under BNF sections **SPECIFICATIONS**, **IMPLEMENTATIONS**, **EXECUTABLES**, **VERIFICATION POINTS**, **VERIFICATIONS**, and **VALIDATIONS**. At the lowest level of the **VALIDATIONS** part of the tree are **ACTIONS** and **CONDITIONS**. All sections will refer to non-terminals described under the **PARAMETERS** and **ATTRIBUTES** sections.

Some terminal symbols include the quote ' symbol. They are used for top level quoted lists. This was done intentionally to make it easy for more advanced work involved with programmatically generating Software Plans as lists and applying the function **SPECIFY-PLAN** to those lists.

TOP LEVEL

```
<plan-specification> ::= (specify-plan {<plan-attribute>}*)
<plan-attribute>
   ::= <plan-name-spec> | <log-file-spec> | <verbose-loading-spec> |
       <undefined-attribute-value-spec> | <collection-coercion-type-spec> |
       <globals-spec> | <specifications-spec> | <implementations-spec> |
       <executables-spec> | <verification-points-spec> |
       <verifications-spec> | <sub-validations-spec> | <validations-spec>
<plan-name-spec> ::= :name '<symbol>
<log-file-spec> ::= :log-file <string>
<verbose-loading-spec> ::= :verbose-loading <choice-symbol>
<undefined-attribute-value-spec> ::= :undefined-attribute-value <host-val>
<collection-coercion-type-spec> ::= :collection-coercion-type '<symbol>
<globals-spec> ::= :globals '({<plan-arg> <global-val>}*)
```

PARAMETERS

```
<global-val> ::= <host-var> | <plan-constant>
<plan-par> ::= <plan-arg> | <recorded-val> | <plan-constant>
<plan-constant> ::= <string> | <integer>
<plan-arg> ::= <symbol>
<host-var> ::= <symbol>
<host-val> ::= <symbol> | <string> | <integer>
<choice-symbol> ::= t | nil
```

ATTRIBUTES

```
<recorded-val> ::= (<object-name> <attribute-name>)
<record-syntax> ::= <record-syntax-1> | <record-syntax-2>
```

```
    <record-syntax-1> ::= <recording-val> <attribute-name>
    <record-syntax-2> ::= <recording-val> <object-name> <attribute-name>
    <recording-val> ::= <verification-form> | <plan-par>
    <object-name> ::= <symbol> | <verification-point-spec>
    <attribute-name> ::= <symbol>

SPECIFICATIONS

    <specifications-spec> ::= :specifications '({<specification-form>}*)
    <specification-form> ::= (<specification-name> {<string>}*)
    <specification-name> ::= <symbol>

IMPLEMENTATIONS

    <implementations-spec> ::= :implementations '({<implementation-form>}*)
    <implementation-form> ::= (<implementation-name> {<string>}*)
    <implementation-name> ::= <symbol>

EXECUTABLES

    <executables-spec> ::= :executables '({executable-form}*)
    <executable-form> ::= (<executable-name> {<plan-par>}*)
    <executable-name> ::= <symbol>



VERIFICATION POINTS

    <verification-points-spec>
        ::= :verification-points '({<verification-point-form>}*)
    <verification-point-form>
        ::= '(<verification-point-name> <verification-point-arg-spec>)
    <verification-point-arg-spec>
        ::= {<point-arg>}* | {<point-arg>}* :output {<point-arg>}*
    <verification-point-name> ::= <symbol>
    <point-arg> ::= <symbol>

VERIFICATIONS

    <verifications-spec> ::= :verifications '({<verification-form>}*)
    <verification-form> ::= '(<verification-name> {<plan-par>}*)
    <verification-name> ::= <symbol>

VALIDATIONS

    <sub-validations-spec> ::= :sub-validations '({<validation-form>}*)
    <validations-spec> ::= :validations '({<validation-form>}*)
    <validation-form> ::= (<validation-name> {<action>}*)
    <validation-name> ::= <symbol>
    <predefined-validation> ::= :report | :log | :abort
```

ACTIONS

```
<action> ::= <validation-reference> | (<action-form> {<condition>}*)
<validation-reference> ::= <validation-name> | <predefined-validation>
<action-form>
  ::= <load-action> | <execute-action> | <engage-action> | <test-action> |
      <record-action> | <collect-action> | <report-action> | <log-action>
<load-action> ::= (:load <load-form>)
<load-form> ::= <specification-name> | <implementation-name>
<execute-action> ::= (:execute <executable-name>)
<engage-action> ::= (:engage :listener)
<test-action> ::= (:test {<verification-name>}*)
<test-spec> ::= <verification-name> | (<verification-name> {<plan-par>}*)
<record-action> ::= (:record <record-syntax>)
<collect-action> ::= (:collect <record-syntax>)
<report-action> ::= (:report <string> {<plan-par>}*)
<log-action> ::= (:log <string> {<plan-par>}*)
```

CONDITIONS

```
<condition> ::= (<condition-form> {<action>}*)
<condition-form> ::= <predefined-condition> | <verification-point-spec>
<predefined-condition> ::= :on-pass | :on-fail | :on-entry | :on-exit
<verification-point-spec>
    ::= (<condition-name> <verification-point-name>)
<condition-name> ::= :before | :after
```

B. Definitions — Terms and Abbreviations

B.1 General Acronyms

ADP	Automatic Data Processing
ADS	Advanced Decision Systems
AFB	Air Force Base
ATD	Acceptance Test Description
ATP	Acceptance/Accreditation Test Plan
ATR	Acceptance Test Report
CC	Command and Control Division
CCB	Configuration Control Board
CDR	Critical Design Review
CDRL	Contract Data Requirements List
CEO	Chief Executive Officer
CFE	Customer Furnished Equipment
CFI	Customer Furnished Information
CFG	Computer Facilities Group
CFS	Customer Furnished Softwar
CI	Configuration Item
CM	Configuration Management
CMP	Configuration Management Plan
COI	Community of Interest
COO	Chief Operating Officer
CPP	Cost Performance Review
CRISD	Computer Resources Integrated Support Document
COTS	Commercially available, Off-the-Shelf
CQAE	Chief Quality Assurance Evaluator
CRC	Cyclic Redundancy Check
CSC	Computer Software Components
CSCI	Computer Software Configuration Item
CSOM	Computer System Operator's Manual
CSP	Communications Support Processor
CSSR	Cost Schedule Status Report
CSU	Computer Software Unit
DCAA	Defense Contracting Audit Agency
DCS	Defense Communications System
DDCMP	Digital Data Communications Message Protocol
DES	Data Encryption Standard
DFAR	Defense Federal Acquisition Regulation
DIA	Defense Intelligence Agency

DID	Data Item Description
DIS	Daily Intelligence Summary
DLR	Direct Labor Rates
DoD	Department of Defense
DSCIS	Daily U.S. Space Command Intelligence Summary Message
DTG	Date Time Group
ECP	Engineering Change Proposal
EPROM	Erasable-Programmable Read-Only Memory
FAR	Federal Acquisition Regulation
FCA	Functional Configuration Audit
FCCM	Facilities Capital Cost of Money
FDMP	Full Duplex Message Protocol
FOC	Final Operational Capability
FQT	Formal Qualification Test
FSM	Firmware Support Manual
GFE	Government Furnished Equipment
GFI	Government Furnished Information
GFS	Government Furnished Software
G&A	General & Administrative
HMI	Human Machine Interface
HOL	Higher order Language
HQ	Headquarters
HWCI	Hardware Configuration Item
ICD	Interface Control Document
ICWG	Interface Control Working Group
IDD	Interface Design Document
IOC	Initial Operational Capability
I/O	Input and/or Output
IP	Internet Protocol
IRS	Interface Requirements Specification
ISS	Intelligence Support Systems
ID	Imagery Division
IV&V	Independent Verification and Validation
I2	Intelligence Data Handling Systems Communications, Version2
JSR	Job Status Report
LAN	Local Area Network
LLCSC	Lower-level computer software component
LOE	Level-of-Effort
MCG	Mapping, Charting and Geodesy
MCCS	Mission-Critical Computer System
MIL	Military

MLDT	Mean Logistics Delay Time
MMI	Man-Machine Interface
MTBF	Mean Time Between Failures
MTBSF	Mean Time Between Software Faults
MTBSH	Mean Time Between Software Halts
MTTR	Mean Time to Repair
NDS	Non-Development Software
NFS	Network File System
NeWS	Network extensible Window System
NLP	Natural Language Processing
OA	Operational Availability
OCD	Operational Concept Document
OCR	Optical Character Reader
OM	Operator's Manual
OS	Operating System
PCA	Physical Configuration Audit
PDR	Preliminary Design Review
PID	Process ID
PM	Program Manager
PQM	Program Quality Manager
PRC	Program Review Coordinator
PROM	Programmable Read-Only Memory
PRT	Program Review Team
QA	Quality Assurance
QAC	Quality Assurance Committee
QAM	Quality Assurance Manager
RCS	Revision Control System
RFP	Request for Proposal
RGB	Red, Green, Blue
ROM	Read-Only Memory
RPC	Remote Procedure Call
RTMS	Real Time Message System
R&D	Research and Development
SA	System Administrator
SAC	Strategic Air Command
SAD	Situation Assessment Division
SCI	Sensitive Compartmented Information
SCIF	Special Compartmented Intelligence Facility
SCN	Specification Change Notice
SDD	Software Design Document
SDF	Software Development Folder
SDL	Software Development Library

SDP	Software Development Plan
SDR	Software Design Review or Software Deficiency Report
SDRL	Subcontract Deliverables Requirements List
SER	Software Engineering Report
SI	System Integrators
SO	Security Officer
SOW	Statement of Work
SPM	Software Programmer's Manual
SQEP	Software Quality Evaluation Plan
SQPP	Software Quality Program Plan
SPS	Software Product Specification
SRR	System Requirements Review
SRS	System Requirements Specification
SSDD	System/Segment Design Document
SSS	System/Segment Specification
STD	Standard
STP	System Test Plan
STR	System Test Report
SU	Superuser
SUM	Software User's Manual
SW	Software
SWCI	Software Configuration Item
TBD	To Be Determined
TCP	Transmission Control Protocol
TCP/IP	Internet protocol suite
TELNET	Telecommunications Network Protocol
TIM	Technical Interchange Meeting
TOMP	Task Order Management Plan
TQM	Total Quality Management
TRR	Test Readiness Review
UDF	Unit Development Folder
UI	User Interface
UM	User's Manual
UPS	Uninterruptible Power Supply
VDD	Version Description Document
V&V	Verification and Validation
WBS	Work Break-Down Structure
WYSIWYG	What You See Is What You Get (pronounced: Whiz-E-Wig)

B.2 Definitions

- **ASCII.** Standard alphanumeric character set for computers.
- **Authentication.** Determination by the Government that specification content is acceptable.
- **Background.** A non-realtime message hardcopy document entered through the optical character readers, magnetic tape, or by hand.
- **Baseline.** A configuration identification document or a set of such documents formally designated and fixed at a specific time during a CI's life cycle. Baselines, plus approved changes from those baselines, constitute the current configuration identification. For 2167A configuration management techniques there are three baselines, as follows:

  1. **Functional baseline.** The initial approved functional configuration identification.
  2. **Allocated baseline.** The initial approved allocated configuration identification.
  3. **Product baseline.** The initial approved or conditionally approved product configuration identification.

  However, for iterative prototyping programs the functional and allocated requirements are not identified until late in the development process. The use of a test or development baseline in place of the functional and allocated baselines, allows for the system evolution. The two standard baselines that ADS utilizes are:

  1. **Test baseline.** The test baseline will be established as segments are released from each task group.
  2. **Product baseline.** The product baseline will be established at the successful completion of contract requirements/deliveries.

- **Baseline Management.** Application of technical and administrative direction to designate the documents which formally identify and establish the initial configuration identification at specific times during its life cycle.
- **Certification.** A process which may be incremental, by which a contractor provides objective evidence to the contracting agency that an item satisfies its specified requirements.
- **Computer data definition.** A statement of the characteristics of the basic elements of information operated upon by hardware in responding to computer instructions. These characteristics may include, but are not limited to, type, range, structure, and value.

- **Computer hardware.** Devices capable of accepting and storing computer data, executing a systematic sequence of operations on computer data or producing control outputs. Such devices can perform substantial interpretation, computation, communication, control, or other logical functions.

- **Computer resources.** The totality of computer hardware, software, personnel, documentation, supplies, and services applied to a given effort.

- **Computer software.** A combination of associated computer instructions and computer data definitions required to enable the computer hardware to perform computational or control functions.

- **Computer Software Component (CSC).** A distinct part of a computer software configuration item (CSCI). CSCs may be further decomposed into other CSCs and Computer Software Units (CSUs).

- **Computer Software Configuration Item (CSCI).** A configuration item for computer software.

- **Computer Software documentation.** Technical data or information, including computer listings and printouts, which documents the requirements, design, or details of computer software, explains the capabilities and limitations of the software, or provides operating instructions for using or supporting computer software during the software's operational life.

- **Computer Software Unit (CSU).** An element specified in the design of a Computer Software Component (CSC) that is separately testable.

- **Configuration.** The functional and/or physical characteristics of hardware/software as set forth in technical documentation and achieved in a product.

- **Configuration control.** The systematic evaluation, coordination, approval, disapproval, and implementation of all approved changes in the configuration of a CI after formal establishment of its configuration identification.

- **Configuration Identification.** The current approved or conditionally approved technical documentation for a configuration item as set forth in specifications, drawings and associated lists, and documents.

- **Configuration Item Number (CIN).** A CIN is a permanent number assigned by the configuration manager to identify a configuration item. The CIN is composed of alpha-numeric characters.

- **Configuration Item (CI).** An aggregation of hardware/software, or any of its discrete portions, which satisfies an end use function and is designated for configuration management.

- **Configuration Mangement.** A discipline applying technical and administrative direction to (1) identify and document the functional and physical characteristics of a configuration items, (2) control changes to those characteristics, and (3) record and report change processing and implementation status.

- **Configuration status accounting.** The recording and reporting of the information that is needed to manage configuration effectively, including a listing of the approved configuration identification, the status of proposed changes to configuration, and the implementation status of approved changes.

- **Cost.** The term "cost" means cost.

  1. **Non-recurring costs.** One-time costs which will be incurred if an engineering change is ordered and which are independent of the quantity of items changed, such as, cost of redesign, special tooling or qualification.

  2. **Recurring costs.** Costs which are incurred for each item changed or for each service or document ordered.

- **Critical Design Review (CDR).** This review shall be conducted for each configuration item when detail design is essentially complete. The purpose of this review will be to (1) determine that the detail design of the configuration item under review satisfies the performance and engineering speciality requirements of the HWCI development specification(s), (2) establish the detail design compatibility among the configuration item and other items of equipment, facilities, computer software personnel, (3) assess configuration item risk areas (on a technical, cost, and schedule basis), (4) assess the resulted of the producibility analyses conducted on system hardware, and (5) review the preliminary hardware product specifications. For CSCIs, this review will focus on the determination of the acceptability of the detailed design, performance, and test characteristics of the design solution, and on the adequacy of the operation and support documents.

- **Critical item.** An item within a CI which, because of special engineering or logistic considerations, requires an approved specification to establish technical or inventory control at the component level.

- **Deficiencies.** Deficiencies consist of two type: (1) conditions or characteristics in any hardware/software which are not in compliance with specified configuration, or (2) inadequate (or erroneous) configuration identification which has resulted, or may result, in configuration items that do not fulfill approved operational requirements.

- **Developmental Configuration.** The contractor's software and associated technical documentation that defines the evolving configuration of a CSCI during development. It is under the development contractor's configuration control and describes the software design and implementation. The Developmental

Configuration for a CSCI consists of a Software Design Document and source code listings. Any item of the Developmental Configuration may be stored on electronic media.

- **Domain.** The area of interest of a particular program.
- **Engineering change.** An alteration in the configuration of a configuration item or item, delivered, to be delivered, or under development, after formal establishment of its configuration identification.
- **Engineering Change Proposal (ECP).** A term which includes both a proposed engineering change and the documentation by which the change is described and suggested.
- **Evaluation.** The process of determining whether an item or activity meets specified criteria.
- **Firmware.** The combination of a hardware device and computer instructions or computer data that reside as read-only software on the hardware device. The software cannot be readily modified under program control.
- **Formal Qualification Review (FQR).** The test, inspection, or analytical process by which a group of configuration items comprising the system are verified to have met specific contracting agency contractual requirements (specifications or equivalent). This review does not apply to hardware or software requirements verified at FCA for the individual configuration item.
- **Formal Qualification Testing (FQT).** A process that allows the contracting agency to determine whether a configuration item complies with the allocated requirements for that item.
- **Functional area.** A distinct group of system performance requirements which, together with all other such groupings, forms the next lower level breakdown of the system on the basis of function.
- **Functional characteristics.** Quantitative performance, operating, and logistic parameters and their respective tolerances. Functional characteristics include all performance parameters, such as range, speed, lethality, reliability, maintainability, and safety.
- **Functional Configuration Audit (FCA).** The formal examination of functional characteristics' test data for a configuration item, prior to acceptance, to verify that the item has achieved the performance specified in its functional or allocated configuration identification.
- **Hardware Configuration Item (HWCI).** A configuration item for hardware.

- **Independent Verification and Validation (IV&V).** Verification and validation performed by a contractor or independent group that is not responsible for developing the product or performing rg w activity being evaluated. IV&V is an activity that is conducted separately from the software development activities.

- **Interface Control.** Interface control comprises the delineation of the procedures and documentation, both administrative and technical, contractually necessary for identification of functional and physical characteristics between two or more configuration items which are provided by different contractors/Government agencies, and the resolution of the problem thereto.

- **Interface Control Working Group (ICWG).** For programs which encompass a system/configuration item design cycle, an ICWG normally is established to control interface activity between contractors or agencies, including resolution of interface problems and documentation of interface agreements.

- **MMI.** Term used to describe the interface between the user, the computer, and the program. Terms include: man-machine interface, human-machine interface, user-interface.

- **Non-development software (NDS).** Software that is not required to be delivered by the contract.

- **Operator.** In text'management, a term that describes the connection between a subtopic and a topic (e.g. and, or not, vote, phrase). For computers, one who maintains the computer.

- **Physical characteristics.** Quantitative and qualitative expressions of material features, such as composition, dimensions, finishes, form, fit, and their respective tolerances.

- **Physical Configuration Audit (PCA).** A technical examination of a designated configuration item to verify that the configuration item "As Built" conforms to the technical documentation which defines the configuration item.

- **Preliminary Design Review (PDR).** This review shall be conducted for each configuration item or aggregate of configuration items to (1) evaluate the progress, technical adequacy, and risk resolution (on a technical, cost, and schedule basis)m of the selected design approach, (2) determine its compatibility with performance and engineering speciality requirements of the HWCI development specification, (3) evaluate the degree of definition and assess the technical risk associated with the selected manufacturing methods/processes, and (4) establish the existence and compatibility of the physical and functional interfaces among the computer software and personnel. For CSCIs, this review will focus on: (1) the evaluation of the progress, consistency, and technical adequacy of

the selected top-level design and test approach, (2) compatibility between software requirements and preliminary design, and (3) on the preliminary version of the operation and support documents.

- **Qualification.**

- **Reusable software.** Software developed in response to the requirements for one application that can be used, in whole or in part, to satisfy the requirements of another application.

- **Software development file/folder (SDF).** A repository for a collection of material pertinent to the development or support of software. Contents typically include (either directly or by reference) design considerations and constraints, design documentation and data, schedule and status information, test requirements, test cases, test procedures, and test results.

- **Software development library (SDL).** A controlled collection of software, documentation, and associated tools and procedures used to facilitate the orderly development and subsequent support of software. The SDL includes the Developmental Configuration as part of its contents. A software development library provides storage of and controlled access to software and documentation in human-readable form, machine-readable form, or both. The library may also contain management adat pertinent to the software development project.

- **Software engineering environment.** The set of automated tools, firmware devices, and hardware necessary to perform the software engineering effort. The automated tools may include but are not limited to compilers, assemblers, linkers, loaders, operating systems, debuggers, simulators, emulators, test tools, documentation tools, and data base management system(s).

- **Software quality.** The ability of a software product to satisfy its specified requirements.

- **Software Specification Review (SSR).** A review of the finalized Computer Software Configuration Item (CSCI) requirements and operational concept. The SSR is conducted when CSCI requirements have been sufficiently defined to evaluate the contractor's responsiveness to and interpretation of the system, segment, or prime item level requirements. A successful SSR is predicated upon the contracting agency's determination that the Software Requirements Specification, Interface Requirements Specification(s), and Operational Concept Document form a satisfactory basis for proceeding into preliminary software design.

- **Specification.** A document intended primarily for use in procurement, which clearly describes the essential technical requirements for items, materials, or services including the procedures by which it will be determined that the requirements have been met.

1. **General specification.** A document which covers the requirements common to different types, classes, grades and/or styles of items or services.
2. **Detail specification.** A document which covers (either within itself or by referencing and supplementing a general specification) the complete requirements for only one type of item, or for a limited number of types, classes, etc. of similar characteristics.
3. **System specification.** A document which states the technical and mission requirements for a system as an entity, allocates requirements to functional areas (or configuration items), and defines the interfaces between or among the functional areas.
4. **Development specification.** A document applicable to an item below the system level which states performances, interface and other technical requirements in sufficient detail to permit design, engineering for service, use, and evaluation.
5. **Product specification.** A document applicable to a production item below the system level which states item characteristics in a manner suitable for procurement, production, and acceptance.

- **Specification Change Notice (SCN).** A document used to propose, transmit, and record changes to a specification.
- **Synonym.** An operator that allows a topic to be a synonym of (the same as) another topic.
- **System.** A composite of subsystems, assemblies (or sets), skills, and techniques capable or performing and/or supporting an operational (or non-operational) role. A complete system includes related facilities, items, material, services, and personnel required for its operation to the degree that it can be considered a self-sufficient item in its intended operational (or non-operational) and/or support environment.
- **System Design Review (SDR).** This review shall be conducted to evaluate the optimization, correlation, completeness, and risks associated with the allocated, technical requirements. Also included is a summary review of the system engineering process which produced the allocated technical requirements and of the manufacturing planning for the next phase of effort.
- **Subcontractor.** A subcontractor is an indeividual, partnership, corporation, or association, who (which) contracts with a contractor to design, develop, design and manufacture, manufacture items, which are or were, designed specifically for use in a military application.
- **System Requirements Review (SRR).** The objective of this review is to ascertain the adequacy of the contractor's efforts in defining system requirements. It will be conducted when a significant portion of the system functional requirements has been established.

- **Software support.** The sum of all activities that take place to ensure that implemented and fielded software continues to fully support the operational mission of the software.

- **Software test environment.** A set of automated tools, firmware device, and hardware necessary to test software. The automated tools may include nbut are not limited to test tools such as simulation software, code analyzers, etc. and may also include those tools used in the software engineering environment.

- **System Specification.** A system level requirements specification.

- **Technical Report.** A technical report encompasses the evaluated relevant facts on a study or phase of a study of a particular art, science, profession, or trade, and stands as a permanent official record in a formal document. The prime purpose of a technical report is to disseminate the results of activity and to foster the exchange of information.

- **Test Readiness Review (TRR).** A review conducted for each CSCI to determine whether the software test procedures are complete and to assure that the contractor is prepared for formal CSCI testing. Software test procedures are evaluated for compliance with software test plans and descriptions, and for adequacy in accomplishing test requirements. At TRR, the contracting agency also reviews the results of informal software testing and any updates to the operation and support documents. A successful TRR is predicated on the contracting agency's determination that the software test procedures and informal test results form a satisfactory basis for proceeding into formal CSCI testing.

- **Unit.** One complete configuration item.

- **Validation.** The process of evaluating software to determine compliance with specified requirements.

- **Vendor.** A vendor is a manufacturer or supplier of a commercial item.

- **Verification.** The process of evaluating the products of a given software development activity to determine correctness and consistency with respect to the products and standards provided as input to that activity.

- **Version.** An identified and documented body of software. Modification to a version of software (resulting in a new version) require configuration management actions by either the contractor, the contracting agency, or both.

- **Waiver.** A written authorization to accept a configuration item or other designated items, which during production or after having been submitted for inspection, are found to depart from specified requirements, but nevertheless are considered suitable for use "as is" or after rework by an approved method.

- **Work Breakdown Structure (WBS).** A product-oriented family tree, composed of hardware, software, services and other work tasks, which results from project engineering effort during the development and production of a defense material item, and which completely defines the project/program. A WBS displays and defines the product(s) to be developed or produced and relates the elements of work to be accomplished to each other and to the end product.

B.3 Document Definitions

- **Configuration Management Plan (CMP).** The Configuration Management Plan (CMP) describes the procedures and methods to be used for configuration management during the life of the program. This include development, testing, and installation.

- **Computer Resources Integrated Support Document (CRISD).** The Computer Resources Integrated Support Document (CRISD) provides the information needed to plan for life cycle support of deliverable software. The CRISD documents the contractor's plans for transitioning support of deliverable software to the support agency.

- **Computer System Operators Manual (CSOM).** The Computer System Operator's Manual (CSOM) provides information and detailed procedures for initiating, operating, monitoring, and shutting down the computer system and for identifying/isolating computer malfunctions.

- **Firmware Support Manual (FSM).** The Firmware Support Manual (FAM) provides the information necessary to load software or data into firmware components of a system. It is equally applicable to read only memory (ROMs), Programmable ROMs (PROMs), Erasable PROMs (EPROMs), and other firmware devices.

- **Interface Control Document (ICD).** The Interface Control Document (ICD) specifies all of the external (other systems) and internal (between subsystems) interfaces necessary to ensure proper development of software for the system. It serves to document and control interface decisions.

- **Interface Design Document (IDD).** The Interface Design Document (IDD) specifies the detailed design for the interface between the CSCIs.

- **Interface Requirements Specification (IRS).** The Interface Requirements Specification (IRS) specifies the requirements for one or more interfaces between one or more CSCIs and other configuration items.

- **Software Data Dictionary Document.** The Software Data Dictionary Document is a technical document prepared for the programmers and data base administrators. It provides for the central collection of information about

all data used by the software system: all files, all record types, all items within records, all relationships between records, and all pertinent information about the use of the data. The Software Data Dictionary Document is designed to provide a standard, consistent, simple framework for information about the data used by the system being developed.

- **Software Design Document (SDD).** The Software Design Document (SDD) describes the complete design of the each CSCI. It describes the CSCI as composed of Computer Software Components (CSCs) and Computer Software Units (CSU's).

- **Software Development Plan (SDP).** The Software Development Plan (SDP) describes a contractor's plans for conducting software development. The SDP is used to provide the Government insight into the organization(s) responsible for performing software development and the methods and procedures to be followed by these organization(s). The SDP is used by the Government to monitor the procedures, management, and contract work effort of the organization performing software development.

- **Software Product Specification (SPS).** The Software Product Specification (SPS) consists of the SDD and source code listings for a CSCI.

- **Software Programmer's Manual (SPM).** The Software Programmer's Manual (SPM) provides information needed by a programmer to understand the instruction set architecture of the specific host or target computers. The SPM provides information that may be used to interpret, check out, troubleshoot, or modify existing software on the host or target computers.

- **Software Quality Program Plan (SQPP).** The Software Quality Program Plan (SQPP) identifies the organizations and procedures to be used by the contractor to perform activities related to the Software Quality Program specified by DoD-STD-2168. The SQPP is used to evaluate the contractor's plans for implementing the Software Quality Program.

- **Software Test Plan (STP).** The Software Test Plan (STP) describes the formal qualification test plans for acceptance testing of the system. The STP identifies the software test environment resources required for accreditation testing. The STP identifies the individual tests that will be performed during accreditation testing.

- **Software Test Description (STD).** The Software Test Description describes each of the procedures identified in the the STP.

- **Software Test Report (STR).** The Software Test Report (STR) is a record of the formal qualification testing performed on the system. The STR provides the Government with a permanent record of FQT performed on the system.

- **Software Users Manual (SUM or UM).** The Software User's Manual (SUM) provides the user personnel with instructions sufficient to run the system.

- **System Operational Concept Document (SOC).** The System Operational Concept Document describes the mission of the system and its operational and support environments. Also described are the functions and characteristics of the computer system within the overall system.

- **System Requirements Specification (SRS).** The Software Requirements Specification (SRS) specifies the engineering and qualification requirements for the system.

- **System/Segment Design Document (SSDD).** The System/Segment Design Document (SSDD) describes the design of the system and its operational and support environments. It describes the organization of the system as composed of Hardware Configuration Items (HWCIs), Computer Software Configuration Items (CSCIs), and manual operations.

- **System/Segment Specification (SSS).** The System/Segment Specification (SSS) specifies the requirements for a system or a segment of a system. The SSS, upon formal approval, becomes part of the Functional (or Test/Development) Baseline.

- **Version Description Document (VDD).** The Version Description Document (VDD) identifies and describes a version of a Computer Software Configuration Item (CSCI) being released.

C. A Guide to CASE Tools

Although this list will quickly become outdated, the impact of Computer-Aided Software Engineering (CASE) of V&V activities is too important to ignore. This manual has argued for the importance of formality in system specification and design. An additional benefit of formality is that formal representations can be manipulated by computers, resulting in better quality than would likely be obtained manually. At a minimum, certain obvious "clerical" errors can be found automatically, such as inconsistencies in interface definitions. Present generation CASE tools are far from ideal, but you may find something in the list below—based on a list compiled by the CASE Research Group of Florida Atlantic University—that will prove very useful on your project.

Adpac Corp. Adpac CASE Tools; 340 Brannan St., San Francisco, CA 94107, 415-974-6699

Advanced Logical Software Anatool; 9903 Santa Monica Blvd., suite 108, Beverly Hills, CA 90212, 213-653-5786

Advanced Technology International, Inc. SuperCase

AGS Management Systems, Inc. Multi/CAM; category: front end; 880 First Ave., King of Prussia, PA 19406, 215-265-1550

American Management Systems, Inc. Life Cycle Productivity System; category: front end, back end; 1777 North Kent St., Arlington, VA 22209, 703-841-6060

Applied Business Technology Corp. Project Workbench; 361 Broadway, New York, NY 10013, 212-219-8945

Applied Data Research, Inc. DEPICTOR; category: front end; Route 206 and Orchard Rd., CN-8, Princeton, NJ 08543

Arthur Andersen & Co. Design/1 (part of Foundation Series); category: front end, back end, RE/M; 33 West Monroe St., Chicago, IL 60603; 69 West Washington, Chicago, IL 60602, 312-580-0069, 312-580-0033, 312-507-5161

Atherton Technology Software BackPlane; 1333 Bordeaux Drive, Sunnyvale, CA, 94089, Tele: 408 734-9822, Fax: 408 744-1607

ASYST Technologies, Inc. The Developer; One Naperville Plaza, Naperville, IL 60540, 800-361-3673

Bachman Information Systems BACHMAN Product Set

Cadre Technologies, Inc Teamwork OS/2 3.0; category: front end; 222 Richmond St., Providence, RI 02903, 401-351-5950, 401-351-CASE

The CADWARE Group, Ltd SYLVA Series; category: Front end

CASET IPSYS Tool Building Kit; 714-496-8670

CaseWare, Inc AMPLIFY; 3530 Hyland Avenue, Suite 115, Costa Mesa, CA 92626, 714-754-0308

The Catalyst Group PATHVU Series; category: RE/M; Peat Marwick Main& Co., 303 East Wacker Dr., Chicago, IL 60601, 800-323-3059, 312-938-5352

CGI Systems, Inc. PACBase, PACBench, PACDesign; category: front end, back end, RE/M; 8200 Greensboro Dr. Suite 1010, McLean, VA 22102, 703-448-8181; 1 Blue Hill Plaza, Pearl River, NY 10965, 914-735-5030

Chen& Associates ER-Designer (ERD); 4884 Constitution Ave, Ste 1E, Baton Rouge, LA 70808, 504-928-5765

Cincom Systems, Inc. Supra, Mantis, Easy PC Contact, CASE Interchange; 2300 Montana Ave., Cincinnati, OH 45211, 800-888-0115

Coding Factory CoFac

Cognos Powercase; 67 S. Bedford St., Burlington, Mass. 01803, 617-229-6600

Computer Associates International, Inc. CA-Datacom, CA-Ideal, CA-Dataquery, CA-Dataquery PC; Computer Associates World Headquarters, 711 Stewart Ave., Garden City, NY 11530, 516-227-3300

Computer Data Systems Scan/COBOL, SuperStructure; 1 Curie Court, Rockville, MD 20850, 202-921-7000

Computer Sciences Corp Design Generator; category: front end; 3610 Fairview Park Dr, Falls Church, VA 22042, 703-876-1000

Computer Systems Advisers, Inc POSE 4.0; 50 Tice Blvd., Woodcliff Lake, NJ 07675, 800-537-4262, 201-391-6500

Compuware Corporation CATI tools: Abend-AID, CICS Abend-AID, CICS RADAR, File-AID family, TransRELATE, PLAYBACK, File PLAYBACK, SIMULCAST, dBUG-AID, XPEDITER, NAVIGATOR; 31440 Northwestern Highway, Farmington Hills, Michigan 48018-5550

Cortex Corp. CorVision, Application Factory; category: front end, back end, RE/M; 138 Technology Dr., Waltham, MA 02154; 100 Fifth Avenue, Waltham, MA 02154-9863, 617-894-7000

Cullinet Software, Inc. IDMS/Architect

D. Appleton Company IDEF/Leverage; 1334 Park View Ave., Suite 220, Manhattan Beach, CA 90266, 213-546-7575

Deft Inc. Deft; 567 Dixon Rd., suite 110, Rexdale, ON M9W 1H7, Canada, 416-249-2246

Deloitte, Haskins& Sells 4Front; 200 East Randolph Dr., Chicago, IL 60601, 312-856-8168

Digital Equipment Corp. DECASE; DECdirect, Continental Blvd., Merrimack, NH 03054, 800-344-4825

ECS Associates SQL-Link-Plus; 3812 Sepulveda Blvd., Torrance, CA 90505, 213-378-9260

ICONIX Software Engineering Inc. PowerTools Series; category: front end, back end, RE/M; 2800 Twenty Eighth St, Suite 320, Santa Clara, CA 90405, 213-458-0092

Forschungszentrum Informatik (FZI) STONE; Haid-und-Neu-Str. 10-14., D-7500 Karlsruhe, Germany, +49-721-6906-731

i-Logix StateMate; 22 Third Ave., Burlington, MA 01803, 617-272-8090

Index Technology Corp. Excelerator 1.84; category: front end; One Main St., Cambridge, MA 02142, 800-777-8858, 617-494-8200

Institute for Information Industry KangaTool Series; category: front-end; 8th Floor, 106 Ho-Ping E. Rd., Taipei, Taiwan, R.O.C.

Integrated Systems, Inc. AutoCode; 2500 Mission College Blvd., Santa Clara, CA 95054, 408-980-1500

Interactive Development Environments Software Through Pictures; category: front end; 595 Market St., 12th Floor, San Francisco, CA 94105, 415-543-0900

KnowledgeWare, Inc. IEW/WS; category: front end; 3340 Peachtree Rd., Atlanta, GA 30026, 404-231-8575, 800-338-4130

Language Technology RECODER, INSPECTOR; category: RE/M; 27 Congress St, Salem, MA 01970, 800-732-6337, 508-741-1507

Learmonth& Burchett Management Systems, Inc. (LBMS) System Engineer (nee Auto-Mate Plus); 1800 West Loop South, Suite 1800, Houston, TX 77027, 713-682-8530, 800-231-7515

Manager Software Products, Inc. Manager Series; category: Front end, back end; 131 Hartwell Ave, Lexington, MA 02173-3126, 617-863-5800

Matterhorn, Inc. HIBOL; category: back end

McDonnell-Douglas ProKit*Workbench STRADIS, PRO-IV; category: front end; P.O. Box 516, Dept. L515, MS 2812301, St. Louis, MO 63166, 800-325-1087, 800-822-7337, 314-232-5715

Mentor Graphics Corp. Analyst/RT, Designer, Auditor; category: front end; 8500 Southwest Creekside Place, Beaverton, OR 97005, 503-626-7000

Meta Systems QuickSpec, Structured Architect (SA), Structured Architect-Integrator (SA-I), PSL/PSA, Report Specification Interface (RSI), View Integration System (VIS); category: front end, RE/M; 315 E. Eisenhower Parkway, Suite 200, Ann Arbor, MI 48108, 313-663-6027

Micro Focus, Inc. COBOL/2 Workbench; 2465 East Bayshore Rd., Palo Alto, CA 94303, 415-856-4161

Netron, Inc. NETRON/CAP; 99 St. Regis Crescent N, Downsview, Ontario, Canada M3J 1Y9, 416-636-8333

On-Line Software International CasePac; 2 Executive Dr., Ft. Lee Executive Park, Ft. Lee, NJ 07024, 201-592-0009

Optima, Inc. DesignVision 1.7, DesignMachine 2.0; category: front end, back end

Oracle Systems Corp. CASE*Designer, CASE*Dictionary, CASE*Generator, SQL*Forms, SQL*Report, SQL*QMX, Oracle, SQL*Louder; Oracle World Headquarters, 500 Oracle Pkwy, Redwood Shores, CA 94065, 415-506-7000; ORACLE Corporation, 20 Davis Drive, Belmont, CA 94002, 800-345-DBMS

Pansophic Systems Inc. Telon; 2400 Cabot Drive, Lisle, IL 60532, 312-505-6000, 800-323-7335

Phoenix Technologies, Ltd. P-Source, P-Tools; 846 University Ave., Norwood, MA 02062, 617-551-4000

Popkin Software& Systems System Architect; 111 Prospect St., Suite 505, Stamford, CT 06901, 203-323-3434

ProMod, Inc. ProMod Series; category: front end, back end, RE/M; 23685 Birtcher Dr., El Toro, CA 92630, 714-855-3046, 800-255-2689

Rational Rational Design Facility; category: front end; 3320 Scott Blvd. Santa Clara, CA 95054

Ready Systems Corp. CardTools; 470 Potrero Ave., P.O. Box 60217, Sunnyvale, CA 94086

Sage Software Inc. Polytron Version Control System (PVCS), APS Development Center; category: back end, RE/M; 1700 N.W. 167th Place, Beaverton, OR 97006, 800-547-4000

Sapiens International Perfect, Object-Modeller, Sapiens, Quix; Sapiens USA, 295 7th Ave., New York, NY 10001, 212-366-9394

Schemacode International Inc Schemacode, Datrix; 89 Gleenbrooke, suite 100, Dollard des Ormeaux, Quebec H9A 2L7, 514-683-8693, fax 514-683-6792, datrix@rgl.polymtl.ca

Six Sigma Case Canonizer; 13456 SE 27th Place, Bellevue, WA 98005, 206-643-6911

Softlab, Inc. Maestro; category: front end, back end, RE/M; 188 The Embarcadero, Bayside Plaza, Suite 750, San Francisco, CA 94105, 415-957-9175

Software AG of North America, Inc. Adabas, Natural, Construct, Predict, Predict Case, Super Natural; 11190 Sunrise Valley Drive, Reston, VA 22091, 703-860-5050

Software Architecture and Engineering Strategic Networked Application Platform; 1600 Wilson Blvd., Arlington, VA 22209, 703-276-7910

StarSys, Inc. MacBubbles; category: front end; 11113 Norlec Dr., Silver Spring, MD 20902

Syscorp International, Inc. MicroStep 1.3; 9420 Research Blvd., Suite 200, Austin, TX 78759, 512-338-0591

Telelogic Europe SDT; 33 Boulevard de la Cambre, B-1050 Brussels, Belgium, 011-32-2-647-3670

Texas Instruments Inc. Information Engineering Facility (IEF) 4.0; 6550 Chase Oaks Blvd., Plano, TX 75023, 800-527-3500

Tom Software Application Xcellence; 127 SW 156th Street, Seattle, WA 98166, 206-246-7022

Tranform Logic Inc. (Previosly Nastec Corp.) DesignAid 4.3; category: front end; 24681 Northwestern Hwy., Southfield, MI 48075, 800-872-8296 7799 Leesburg, Suite 1110, North Tower, Falls Church, VA 22043, 703-556-9401

Transform Logic Corporation Transform; 8502 East Via de Ventura, Scottsdale, AZ 85258, 602-948-2600

Unisys Corp. Linc Design Assistant, Linc, Mapper, DMS II; P.O. Box 500, Bluebell, PA 19424, 215-986-4011

ViaSoft, Inc. Via/Insight, Via/SmarTest; 3033 North 44th St., Suite 280, Phoenix, AZ 85018, 602-952-0050

Visible Systems Corp. Visible Analyst Workbench; category: front end; 950 Winter St., Waltham, MA 02154, 617-969-4100

Visual Software, Inc. vsDesigner, vsSQL, vsObject Maker; category: front end; 3945 Freedom Circle, Suite 540, Santa Clara, CA 95054, 408-988-7575

Westmount Technology B.V. ISEE, TSEE, RTEE; 5020 148th Ave. N.E., P.O. Box 97002, Redmond, WA 98073-9702

Yourdan, Inc. Analyst/Designer Toolkit, Cradle; category: front end; 1501 Broadway, New York, NY 10036, 212-391-2828

MISSION

OF

ROME LABORATORY

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C^3I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.